ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación:

INGENIERO TÉCNICO EN INFORMÁTICA DE GESTIÓN

Título del proyecto:

MEJORA E IMPLEMENTACIÓN DE UN SISTEMA DE GESTIÓN DE INVENTARIO PARA EL LABORATORIO DE FOTÓNICA DE LA UNIVERSIDAD PÚBLICA DE NAVARRA

Xabier-Cernin Napal Roncal

Santiago Tainta Ausejo

Pamplona, Septiembre 2016

Índice

Índice de figuras

1. Objetivos

El **Laboratorio de Fotónica**, perteneciente al Departamento de Ingeniería Eléctrica y Electrónica de la **Universidad Pública de Navarra**, cuenta en la actualidad con un *software* para la gestión y organización del material disponible en el mismo. Este sistema es ampliamente utilizado por los integrantes del departamento, y aunque cumple con su función, se encuentra desfasado y no cuenta con todas las herramientas que demandan los usuarios de la aplicación, ya que se realizó como un parte de un Proyecto Final de Carrera hace seis años.

Se propone como *PFC* el desarrollo e implantación de una nueva versión de esta plataforma, con el fin de intentar mejorar la usabilidad y sus características, para así facilitar a los usuarios del laboratorio la búsqueda y uso del material disponible. El objetivo del departamento es proveer a sus usuarios de un sistema sencillo, accesible vía web desde cualquiera de los equipos disponibles en la red interna de la Universidad, que facilite el día a día a todo aquél estudiante, profesor o integrante de la Universidad el uso del material y los elementos disponibles en este almacén.

En todo momento el sistema ha de garantizar la integridad y seguridad de todos los datos que almacene, siendo información crítica para este departamento, ya que no sólo se utilizará como medio para el inventariado el almacén, sino que permitirá almacenar documentación digital y gestionar el uso de los recursos por parte de los usuarios. Además del desarrollo de la aplicación, será necesario realizar una migración de la información de la información disponible en el sistema actual, siendo clave asegurar que no se produzca una pérdida de información durante el proceso.

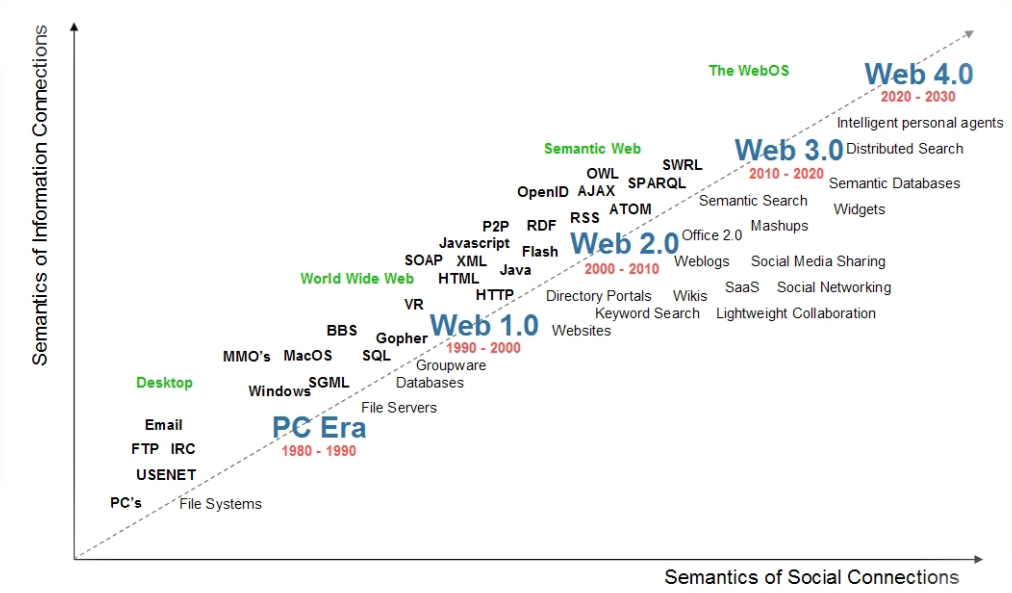
Este es un desarrollo muy adecuado para un Proyecto Final de Carrera ya que integra todos los aspectos que un Ingeniero Informático puede llegar a necesitar dominar durante su vida laboral, requiriendo conocimientos de los siguientes aspectos:

- *System administrator*: campo responsable de la implementación, configuración y mantenimiento de un sistema informático, gestionando el uso de sus recursos, asegurando su correcto funcionamiento y garantizando la seguridad del mismo.
- *Database administrator*: área encargada de la administración y gestión de una base de datos, siendo capaz de planificar y desplegar complejos sistemas para el almacenamiento de datos asegurando su integridad.
- *Front-end developer*: consiste en el desarrollo de arquitecturas web utilizando herramientas capaces de ejecutarse de forma nativa en navegadores web, siendo necesarios conocimientos en tecnologías como *HTML*, *CSS* y *JavaScript*.
- *Back-end developer*: área encargada del desarrollo y mantenimiento del núcleo lógico de cualquier *software*, siendo necesario un dominio completo de varios idiomas de programación.

2. Estado del arte

La invención de la **World Wide Web** en 1989 supuso un cambio de mentalidad en la forma de trabajar con ordenadores. La posiblidad de conectar e interactuar entre usuarios de diferentes equipos creó un nuevo mundo de tecnologías para optimizar procesos colaborativos y compartir información.

Las tecnologías web han evolucionado de forma notable en los últimos diez años, llegando a convertirse en estándares de facto en la mayoría de sistemas desarrollados actualmente. La razón principal de ello sea probablemente la posibilidad de alcanzar a un público muy amplio gracias a sus capacidades multiplataforma y multidispositivo, ya que sólo es necesario un navegador web para poder acceder a ellas.



Fuente: Radar Networks & Nova Spivack, 2007
Distribuido bajo licencia **Creative Commons**

Figura 2.1: Evolución de las tecnologías web

Como se puede ver en el gráfico anterior, el mundo web ha ido creciendo cada vez más. Las sencillas páginas estáticas que no permitían ninguna interacción se han ido transformando en completos sistemas y aplicaciones totalmente dinámicas.

La evolución de la web no sólo ha afectado al usuario final en su forma de entenderla y utilizarla. A la hora del desarrollo también han aparecido multitud de sistemas y plataformas destinadas a facilitar la implementación y actualización de las aplicaciones. Desde la evolución de los idiomas de programación hasta la aparición y popularización de *frameworks*, *plugins* y herramientas, hoy en día es posible crear aplicaciones web mucho más completas, accesibles e intuitivas de forma rápida y sencilla.

3. Sistema actual

Actualmente el Departamento de Ingeniería Eléctrica y Electrónica dispone de una solución web para la gestión del almacén de Laboratorio de Fotónica. Sin embargo, ésta se desarrolló en el año 2010, cuando las tecnologías web modernas aún no estaban consolidadas en el mercado. Por este motivo, la interfaz de usuario es excesivamente compleja y poco intuitiva, dificultando su usabilidad. Unido a esto, en el momento del desarrollo del sistema no se contemplaron las posibles necesidades futuras de los usuarios, siendo excesivamente complejo añadir nuevas funcionalidades a la aplicación.

Por último, y no menos importante, existen numerosos *bugs* en el sistema, tanto internos como en la interfaz, siendo algunos altamente críticos y pudiendo derivar en posibles pérdidas o corrupciones de datos. Se detallan los más notables a continuación:

- El sistema de búsqueda de elementos no funciona correctamente. A pesar de que el número de entradas no es excesivamente elevado (en el momento de escribir estas líneas existen aproximadamente 600 entradas), buscar un componente concreto de forma manual es una tarea tediosa, ya que no es posible visualizar más de un elemento de forma simultánea.
- Existe un *bug* de tipo **race condition**[1] crítico a la hora de añadir una entrada nueva al sistema. En caso de que distintos usuarios intenten añadir entradas al sistema, el identificador único de cada una de ellas no se autoincrementará correctamente, sobreescribiendo los datos de todos ellos excepto uno.
- La mayor parte de áreas de la aplicación son vulnerables a ataques de **SQL injection**[2] y **cross-site scripting**[3], por lo que, entre otras cosas, el sistema de credenciales de usuarios, que limita las acciones a realizar en base a permisos de lectura, escritura y administración, se vuelve ineficaz. Además, las contraseñas de los usuarios se encuentran almacenadas en texto plano sin encriptar, siendo muy sencillo obtenerlas directamente de la propia aplicación.

Afortunadamente, ya que el número de usuarios simultáneos del sistema no suele ser elevado, unido a que la aplicación sólo es accesible de forma interna, no ha habido constancia de que estos fallos críticos se hayan producido.

Finalmente, es destacable que no sólo la aplicación utiliza tecnología obsoleta. El sistema operativo sobre el que se ejecuta, *Windows XP*, también está desfasado, estando desde 2014 sin soporte oficial.

[1] condición que se produce cuando múltiples procesos intentan realizar simultáneamente la misma tarea, y el resultado depende del orden de ejecución.

[2] método de ataque capaz de inyectar código *SQL* malicioso en consultas a una base de datos, originado en un fallo a la hora de filtrar información introducida por el usuario susceptible de utilizarse como parámetro para una consulta.

[3] fallo de seguridad en aplicaciones web, que permite inyectar código de forma temporal o permanente, en forma de *scripts JavaScript* en la interfaz de una página web visitada por el usuario.

4. Requisitos

Se desarrollará una aplicación web, accesible desde la red interna de la Universidad, para categorizar y gestionar el inventario del Laboratorio de Fotónica. Será compatible en sus funcionalidades y en su interfaz con los navegadores y sistemas operativos utilizados por los usuarios del Departamento de Ingeniería Eléctrica y Electrónica, siendo obligatorio el desarrollo de una interfaz *responsive*[4]. Deberá funcionar en un entorno virtualizado con *KVM* [1] y *Proxmox* [2] dentro de la red de la Universidad.

Los usuarios de la aplicación se clasificarán según su permiso de acceso. Existirán tres niveles de acceso: usuarios de sólo lectura, usuarios con permisos de escritura, con potestad para añadir, editar y eliminar entradas del inventario, y usuarios administradores. Éstos últimos podrán gestionar la aplicación por completo, pudiendo añadir nuevos usuarios y modificar la información almacenada en el sistema.

En caso de que un usuario no recuerde su contraseña, debe de existir un método por el cuál el usuario sea capaz de modificar su contraseña sin que sea necesaria la intervención de un administrador en este proceso.

Los elementos del inventario se clasificarán siguiendo una estructura de categorías con múltiples subniveles. A la hora de crear o modificar un elemento, éste ha de pertenecer obligatoriamente a una categoría de cualquier nivel existente. Además de los campos comunes a todos los elementos, cada categoría puede contar con campos exclusivos. Todos estos campos han de ser gestionables desde el panel de administración.

El sistema ha de contar con la posibilidad de incluir una imagen para cada elemento, así como toda clase de ficheros digitales asociados a una entrada. Es necesario desarrollar un método sencillo, utilizando, cuando el navegador lo permita, la técnica *drag & drop*, para permitir al usuario la subida de cualquier archivo a la aplicación.

Es necesario desarrollar un sistema de búsqueda que permita encontrar fácilmente cualquier entrada a través de una búsqueda por palabras clave. También se requiere de la existencia paralela de otro sistema de búsqueda más completo, que permita filtrar los elementos en base a la combinación de cualquiera de sus campos.

El panel de administración de la aplicación ha de permitir gestionar tanto el área de inventario (control del sistema de categorías y personalización de la información disponible en las entradas) como el área de usuarios (administración de los permisos de acceso de los usuarios y sus privilegios), así como la realización de copias de seguridad y administración de las actualizaciones del sistema operativo. El administrador de la aplicación también ha de poder gestionar el envío de mensajes a los usuarios de la misma, a través de un sistema de notificaciones.

Finalmente, la aplicación contará con soporte multiidioma, estando disponible en español e inglés, aunque adaptable en un futuro a otros idiomas.

[4] diseño web adaptable al tamaño de pantalla y características del dispositivo utilizado, ya sea ordenador, teléfono móvil y *tablet*.

5. Tecnología

Todas las tecnologías utilizadas se caracterizan por ser *Open Source* y por contar con una comunidad de usuarios y desarrolladores muy amplia.

> *"The idea of copyright did not exist in ancient times, when authors frequently copied other authors at length in works of non-fiction. This practice was useful, and is the only way many authors' works have survived even in part."*
>
> — Richard Stallman (*rms*), *fundador de la Free Software Foundation*

5.1. Sistema operativo

A la hora de elegir un sistema operativo, el primer paso es la elección de la plataforma a utilizar. Existen multitud de plataformas diferentes, entre las que destacan **Windows**, **Linux** y **BSD**, tanto por sus capacidades como por su alto uso en el mercado.

Uno de los requisitos de la aplicación es que debe poder de funcionar en un entorno con pocos recursos. La versión para servidores de **Windows**, a pesar de ser potente y ofrecer numerosas herramientas, es la que más recursos consume. Además, es necesaria una licencia con alto coste económico para utilizarla legalmente y es software totalmente cerrado y privativo.

BSD, al igual que *Linux*, se basa en el sistema operativo *Unix*. Sin embargo, el soporte de *hardware* y la comunidad de desarrolladores es inferior a la de *Linux*. Gran cantidad del software disponible para esta plataforma son *ports* de aplicaciones *Linux*, la mayoría desarrolladas por usuarios externos a los proyectos oficiales y con un soporte y un ritmo de actualizaciones mucho menor.

La plataforma de código abierto **Linux** [3] (o de forma más precisa, *GNU/Linux*[5]) es la solución más óptima para este desarrollo. Su arquitectura modular, su bajo consumo de recursos y su amplia comunidad de usuarios y programadores, permiten una gran flexibilidad en el desarrollo, ejecución y mantenimiento de aplicaciones, así como la posibilidad de ejecutarlas en un *hardware*, tanto físico como virtual, con recursos limitados y sin problemas de compatibilidad de dispositivos.

Dentro del ecosistema *Linux*, existen gran cantidad de distribuciones, sistemas basados en el *kernel* Linux con un conjunto de herramientas, utilidades y características creadas para suplir diferentes necesidades. Dos de las distribuciones más importantes son **Debian** [4] y **CentOS** [5], habiéndose optado por *CentOS* al ser una distribución orientada especialmente para su uso en servidores. Es conocida por ser estable, fiable y

[5]Linux es únicamente el núcleo del sistema operativo, basado en *Unix*, mientras que un importante conjunto de herramientas utilizadas pertenecen al proyecto GNU.

robusta, al estar basada en la distribución **RHEL** [6] (*Red Hat Enterprise Linux*), la distribución comercial de *Linux* más importante. Además, el soporte y mantenimiento de cada una de sus versiones se prolonga durante 10 años (la versión utilizada en este proyecto, *CentOS 7*, tiene soporte hasta 2024), y el número de paquetes y utilidades orientadas al desarrollo y mantenimiento de servidores es lo suficientemente extenso como para realizar cualquier tipo de tarea.

5.2. Back-end: el motor de la aplicación

Tradicionalmente, la infraestructura predominante en el desarrollo web ha sido el *stack* basado en **xAMP** (*Apache* como servidor web, *MySQL* como gestor de bases de datos y *PHP* como lenguaje de programación de lado de servidor). Incluso hoy es, probablemente, la forma más utilizada para implementar cualquier servicio web. Sin embargo, previamente al comienzo de un desarrollo es una buena práctica elegir la herramienta adecuada para la tarea a realizar, por lo que se analizarán cada una de las áreas de este *stack* y se buscará una alternativa más acorde.

> *"About the use of language: it is impossible to sharpen a pencil with a blunt axe. It is equally vain to try to do it with ten blunt axes instead."*
>
> — Edsger Dijkstra, *científico de la computación*

5.2.1. Lenguaje de programación

La correcta elección del lenguaje de programación utilizado influirá directamente en las capacidades y posibilidades de la aplicación a desarrollar. Existen multitud de idiomas que pueden utilizarse en entornos web, siendo los más conocidos y usados **PHP**, **Python** y **Node.js**. Existen otros lenguajes que permiten el desarrolo de entornos web, como **Ruby**, pero han sido descartados por ser más complejos de aprender, desarrollar y mantener. Esto ha afectado seriamente tanto a su cuota de mercado como a la cantidad de documentación disponible.

PHP [7] posee una potente comunidad con años de experiencia, al ser uno de los primeros lenguajes de lado de servidor utilizados en el desarrollo web. Además, existe gran cantidad de documentación que permite que incluso programadores principiantes puedan desarrollar una aplicación de forma rápida y sencilla. Sin embargo, ésto es también un arma de doble filo, ya que siguiendo estos manuales es muy fácil crear código ineficiente o con fallos de seguridad. Por otro lado, la estructura y forma de programar hace que sea muy fácil acabar creando *código spaghetti*[6] que, a largo plazo, hace que la aplicación no sea sencilla de mantener.

Node.js [8] es una tecnología novedosa que en los últimos años ha ganado mucha popularidad. Permite desarrollar servicios web rápidamente, sin necesitar de muchos

[6] término utilizado para definir aquellos programas cuya estructura es muy compleja e insostenible.

recursos de *hardware*. Sin embargo, trabaja normalmente en un nivel de abstracción más bajo que otros idiomas, como es el caso a la hora de gestionar las conexiones *HTTP* directamente a través de los *sockets* del sistema. Además, requiere extensos conocimientos sobre los patrones de programación basados en *callbacks*, *futures* y *promises*, y necesita dedicar muchos esfuerzos en tratar y controlar todas las posibles *excepciones* que puedan producirse.

Python [9], por su flexibilidad como lenguaje multiuso y su gran legibilidad, gracias a características como la *indentación* obligatoria y la ausencia de llaves en su notación, es el idioma idóneo, tanto para escribir código robusto de forma rápida como para realizar futuras modificaciones. Es un lenguaje muy popular y en gran expansión hoy en día, siendo al mismo tiempo muy maduro y fiable gracias a su larga trayectoria y el gran número de colaboradores que hay detrás de él.

Al ser un lenguaje multiparadigma, permite una gran flexibilidad a la hora de elegir una metodología de programación, y cuenta con una librería estándar con gran cantidad de funcionalidades nativas. Además, es un lenguaje multiplataforma y se considera que es uno de los idiomas que más potencian la productividad del desarrollador. Finalmente, la existencia de *frameworks* orientados al desarrollo web y muy sencillos de utilizar, junto a la gran cantidad de librerías disponibles para facilitar el desarrollo, hacen que **Python 3.5**, la última versión disponible, sea el lenguaje elegido para este proyecto.

5.2.2. Gestor de bases de datos

El motor de bases de datos relacionales más popular en entornos web es **MySQL** [10], habiendo ganado popularidad en los últimos años su *fork*[7] **MariaDB** [11]. A la hora de realizar aplicaciones sencillas estos motores cumplen su función, sin embargo, no cuentan con muchas de las características más avanzadas presentes en otros motores.

Para el desarrollo de esta aplicación se utilizará el motor **PostgreSQL** [12]. Es uno de los primeros motores de bases de datos relacionales desarrollados, por lo que su estabilidad y potencia están garantizadas. Además, cuenta con un gran número de funciones avanzadas no disponibles en otros motores, que permiten la implementación de ciertas funcionalidades que de otra manera serían más costosas y complejas. A lo largo del desarrollo de la aplicación se utilizarán principalmente dos de las herramientas avanzadas que provee el motor de *PostgreSQL*: la búsqueda *full-text* y los tipos de datos *JSON*.

Existen también otro tipo de bases de datos que no utilizan esquemas relacionales. Este conjunto de bases de datos, conocidas como bases de datos *NoSQL*, que permiten almacenar la información en formas diferentes al tradicional formato de tablas y filas, como por ejemplo, las bases de datos documentales, de grafos o de clave/valor. Para almacenar los datos *JSON* se podría haber optado por utilizar una base de datos alternativa que siguiese un esquema documental, como **MongoDB** [13]. Sin

[7] proyecto de software clonado de un paquete original, pero desarrollado de forma independiente.

embargo, se ha preferido no hacerlo, ya que al utilizar dos gestores diferentes el nivel de complejidad de la aplicación sería más alto, aumentando la dificultad a la hora de mantener intacta la integridad de los datos entre los dos sistemas.

5.2.3. Servidor web

Desde hace más de 20 años, el servidor **Apache** [14] ha sido el más utilizado en Internet, siendo en gran medida responsable del crecimiento y auge de la red de redes. Aún a día de hoy es utilizado en más del 50 % de servidores web de todo el mundo. El mayor punto en contra de *Apache* es su estructura monolítica y su forma de tratar las peticiones entrantes, creando un proceso independiente para cada una de ellas. Esto hace que requiera muchos recursos en momentos de alta carga de trabajo.

El servidor web **Nginx** [15] se diseñó con la idea de ser mucho más eficiente en escenarios de alta concurrencia, siendo capaz de gestionar múltiples peticiones de forma simultánea sin necesidad de instanciar procesos nuevos constantemente. Además, su arquitectura en forma modular hace que sea incluso aún más ligero.

A pesar de que esta aplicación no se caracteriza por tener una carga de trabajo concurrente elevada, se ha elegido utilizar *Nginx* ya que incluso en escenarios de menor magnitud consume muy pocos recursos, y es capaz de servir peticiones estáticas de forma mucho más rápida que su directo competidor *Apache*.

A lo largo de los años se han ido desarrollando multitud de técnicas y herramientas para poder servir contenido dinámico como respuesta a una petición web. Originalmente se desarrolló el protocolo **CGI** [16] (*Common Gateway Interface*), el cual instanciaba en cada petición un proceso externo que recibía el contenido de la petición y enviaba una respuesta al servidor, quien a su vez la enviaba al cliente. Esta tecnología ha ido evolucionando con los años, y en la actualidad existe una interfaz denominada **WSGI** [17], desarrollada pensando para usarse principalmente con *Python* como lenguaje, y la cual implementa múltiples mejoras de diseño al protocolo original. Por ejemplo, no requiere instanciar un nuevo proceso por cada petición, y es capaz de parsear previamente las cabeceras enviadas por el usuario. Esta será la tecnología utilizada para unificar el servidor y la aplicación, utilizando un software llamado **uWSGI** [18].

5.2.4. Framework

Desarrollar una aplicación web desde cero puede ser una tarea compleja debido a la gran cantidad de funcionalidades que requiere, tales como enrutado de direcciones, sistemas de autenticación, validación de formularios o conexiones a la base de datos. La cantidad de código necesario para todo ello y el tiempo invertido en el desarrollo pueden multiplicar varias veces el coste del desarrollo final. Por ello se utilizan *frameworks*, módulos que proveen de todas las herramientas necesarias a lo largo del desarrollo.

Existen multitud de *frameworks* para *Python*, siendo **Django** [19] probablemente el

más utilizado. Sin embargo, tiene multitud de funcionalidades que no son necesarias para este proyecto, por lo que su complejidad y su curva de aprendizaje son mayores, además de ser algo más lento que otros *frameworks* más ligeros.

Para este proyecto se utilizará **Flask** [20], un *framework* muy ligero y muy rápido, muy utilizado en desarrollos más pequeños y que no requieren de funcionalidades muy complejas. Este *framework* utiliza el patrón de diseño **MVC** (*Modelo-Vista-Controlador*), en el cual los datos, la lógica y la interfaz de usuario se separan en tres capas independientes:

- *Modelo*: es la representación abstracta de los datos utilizados por la aplicación, y el encargado de acceder y modificar dichos datos.
- *Vista*: es la interfaz de usuario, presenta los datos (modelo) solicitados por el usuario. En Flask, éstas se denominan *templates*.
- *Controlador*: es el encargado de responder a las peticiones del usuario, obtener los datos del modelo necesarios y enviarlos a las vista para mostrarlos por pantalla. Para Flask, los controladores son *views*, nombre que puede confundirse con el de las vistas.

Por si solo, *Flask* únicamente provee del enrutado y mapeo de *URLs* a funciones y de un sistema de *templates* para las vistas, **Jinja2** [21]. Para añadir otras funcionalidades es posible aprovechar sus capacidades modulares, instalando varios *plugins*:

- **Flask-SqlAlchemy** [22]: extensión para simplificar el uso de *SqlAlchemy* [23], herramienta para gestionar el manejo de consultas a la base de datos.
- **Flask-Login** [24]: *plugin* que provee de las funciones básicas para gestionar la autenticación y los permisos de usuario.
- **Flask-WTF** [25]: herramienta que integra la funcionalidad del *plugin WTForms* [26] encargado del manejo y validación de formularios completados por el usuario.
- **Flask-BabelEx** [27]: *plugin* que permite añadir las funcionalidades multi-idioma de internacionalización y localización en la aplicación.

5.3. Front-end: la interfaz de usuario

Para el desarrollo de la interfaz de cliente también se va a hacer uso de un *framework* especializado para simplificar la implementación de todas las herramientas visuales, tanto de su diseño como de sus herramientas. En este caso se utilizará un *framework* llamado **Bootstrap** [28], desarrollado por el equipo técnico de *Twitter*. Este es un *framework* muy popular en la actualidad gracias a la cantidad de herramientas que provee para desarrollar fácilmente interfaces web completas y con utilidades *responsive*.

Gracias al uso tan extendido de este *framework*, existe una gran comunidad de desarrolladores a su alrededor, por lo que la cantidad de *plugins* y *templates* disponibles es muy elevada. En este proyecto se utilizará un template conocido como **Gentelella** [29], que provee de una interfaz sencilla y limpia, la cual cumple con creces todas las necesidades visuales de la aplicación.

Uno de los requisitos de la aplicación es que sea posible su ejecución de forma fluida en todo tipo de dispositivos. Hoy en día, utilizando navegadores actuales y equipos de última generación, no es fácil observar bajadas de rendimiento por un exceso de *scripts* de lado de cliente, aunque en equipos de menores prestaciones y dispositivos móviles, el abuso de estos *scripts* puede afectar seriamente a la velocidad de navegación. Por tanto, se procurará reducir al mínimo las animaciones gráficas y los *scripts JavaScript* a ejecutar, limitando su uso a la creación de elementos visuales como *popups* o desplegables, así como a ciertos aspectos funcionales del panel de administración.

La primera elección tomada para evitar el abuso de ejecución de código en el navegador del cliente es evitar el uso del *framwork* **jQuery** [30]. Éste *framework*, muy popular en el desarrollo de aplicaciones web, facilita muchas de las tareas de modificación del *DOM*[8], además de contar con multitud de herramientas básicas para el programador. Sin embargo, el hecho de añadir una capa más en el código hace que la velocidad del mismo se vea afectada notablemente.

No incluir *jQuery* entre los *scripts* disponibles en la aplicación supone un problema, ya que *Bootstrap* hace un amplio uso del mismo para varias de sus funcionalidades. Por ello, será necesario reescribir parte del código del *framework* para hacerlo compatible con *JavaScript* nativo, para así poder hacer uso de varias de sus herramientas, como los diálogos modales.

Finalmente, se utilizarán dos librerías muy ligeras *JavaScript* para implementar algunos aspectos visuales de la aplicación, aunque, de nuevo, se reducirá su uso todo lo posible. Éstas son **Opentip** [31] para la gestión de *tooltips*[9] de ciertos elementos gráficos, y **clipboard.js** [32] para gestionar de forma sencilla el copiado de información al portapapeles del sistema.

[8] *Modelo de Objetos del Documento*: *API* que define la estructura de un documento *HTML* y permite el acceso y manipulación de cualquiera de sus elementos.

[9] herramientas visuales mostradas al situar el cursor del ratón sobre algún elemento, mostrando un mensaje de ayuda para informar de su finalidad.

6. Estructura

6.1. Estructura del sistema operativo

Para optimizar al máximo el funcionamiento de la aplicación y evitar en la medida de lo posible cualquier ralentización del sistema, se utilizará una versión *minimal* de la distribución **CentOS 7** para arquitectura **x64**, al ser la única arquitectura con soporte oficial. Las distribuciones *minimal* se caracterizan por incluir únicamente el *kernel* y los paquetes necesarios para que el sistema operativo sea funcional.

Al instalar la distribución, sólo se encuentra disponible un servidor *SSH* (*Secure-SHell*) para acceder de forma remota al sistema. Por tanto, será necesario instalar el servidor web, el gestor de bases de datos, el intérprete del lenguaje de programación y un servidor *FTP*, además de múltiples paquetes necesarios en diferentes áreas del desarrollo, los cuales se irán enumerando a lo largo de la memoria, según sean necesarios. El número de servicios y programas en ejecución del sistema será mínimo, ya que de esta forma se mejorará el rendimiento del mismo y se reducirá el mantenimiento, al no depender de actualizaciones para corregir *bugs* asociados a programas externos.

Finalmente, se configurarán medidas de seguridad para controlar y restringir los accesos no autorizados. Se instalará un *firewall* y se configurarán las políticas de seguridad del módulo de sistema **SELinux** [33] para permitir a los servicios disponibles acceder a los recursos necesarios para el correcto funcionamiento de la aplicación.

6.2. Estructura de la base de datos

6.2.1. Requisitos funcionales

La base de datos ha de almacenar el listado de usuarios de la aplicación. De cada usuario es necesario almacenar su nombre de usuario y su contraseña codificada, utilizados para acreditarse en el sistema, así como su nombre, apellidos, DNI y correo electrónico. También se indicarán los permisos de acceso que posee, divididos en permisos de acceso, lectura y escritura, y el idioma en el que se mostrará la aplicación una vez acreditado. Cada usuario pertenece a un tipo o categoría, como pueden ser estudiante, profesor o doctorado. Los tipos de usuarios no poseen otros datos, y serán definidos manualmente por el administrador desde la interfaz de la aplicación.

El sistema de inventario está dividido en categorías, las cuales se estructuran en forma de árbol con múltiples subcategorías anidadas en un número indeterminado de niveles. De cada categoría se guardará su nombre y el listado de sus campos dinámicos. Los elementos del inventario han de tener como mínimo un identificador único, un nombre y una fecha de entrada. Además, pueden contar con un número de serie, una descripción y comentarios, así como campos cuyos valores serán definidos por los administradores: estado (OK, averiado...) y ubicación de su documentación (armario, archivador...).

Los usuarios pueden hacerse responsables de varios elementos, así como usarlos en todo momento. En caso de ser utilizados, es posible indicar la fecha de comienzo de uso, la fecha estimada de fin y el motivo de su uso.

Cada una de las entradas del inventario está asociada a una única categoría, pudiendo tener una categoría ninguna, una o varias entradas. Es posible que existan entradas sin categoría asociada, en caso de borrar una categoría no vacía. Además, pueden contener diferentes campos dinámicos gestionados desde la propia aplicación, divididos en dos áreas: los comunes a todas las entradas y aquellos relacionados con la categoría del elemento. Estos campos se almacenarán de forma dinámica.

6.2.2. Modelo entidad-relación

No es posible representar por completo el modelo E-R ya que los campos dinámicos se almacenarán de forma no relacional. En la entidad **Entry** existirán dos atributos, *fields* y *category_fields*, para representar estos documentos. Además existe una entidad **EntryField** que no se encuentra en el diagrama ya que se utilizará de forma dinámica, tal como se explica en la sección **6.2.5** (**Datos no relacionales**).

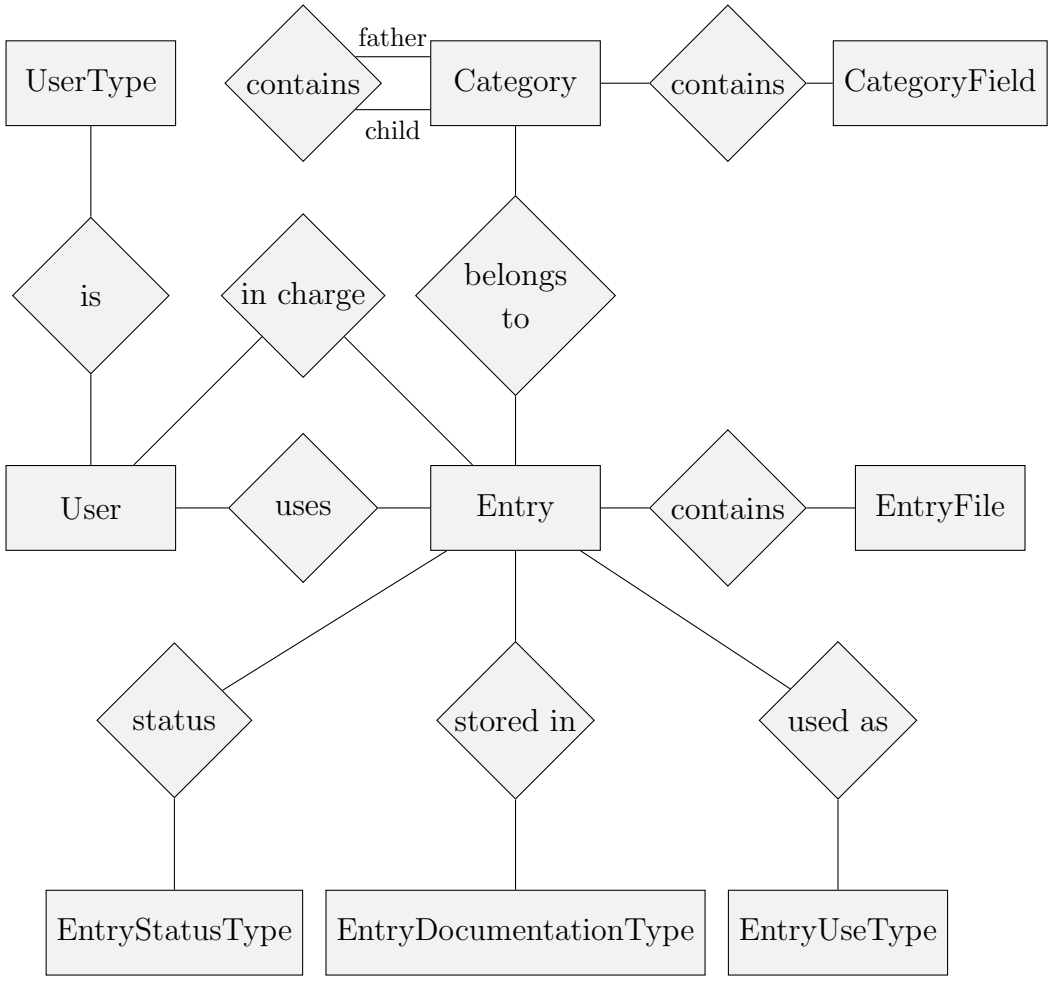


Figura 6.1: Modelo entidad-relación de la base de datos

6.2.3. Modelo relacional

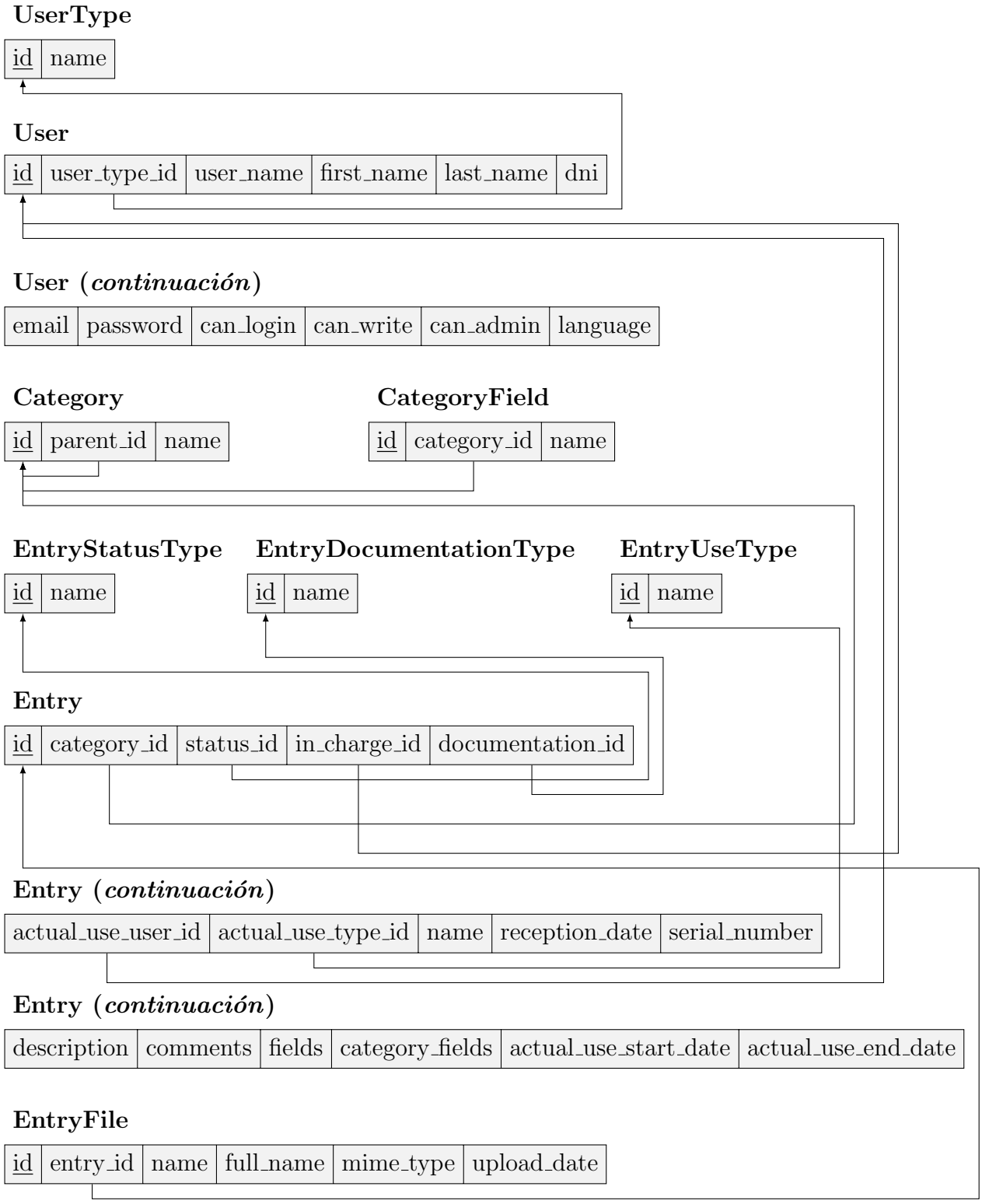


Figura 6.2: Modelo relacional de la base de datos

6.2.4. Paso a tablas

A pesar de ser el último paso lógico en la creación de una base de datos relacional, en este proyecto no será necesario, gracias a la utilización de una técnica llamada **mapeo objeto-relacional**, que permite convertir los modelos de la aplicación, creados en base al modelo relacional, en las tablas necesarias de la base de datos. Esta técnica se explicará detalladamente en el apartado **8.4** (**Base de datos**).

6.2.5. Datos no relacionales

Existen dos tipos de campos dinámicos diferentes: los globales, comunes a todas las entradas, y los específicos de cada categoría. Para almacenarlos se utilizan dos tablas en la base de datos: **EntryFields** y **CategoryFields**. Ambas contienen un conjunto de identificadores únicos con un nombre asociado para el campo.

Para almacenar los campos dinámicos en cada una de las entradas se utilizarán objetos **JSON**, relacionando los identificadores únicos de cada campo con sus datos correspondientes y asociándolos como claves y valores.

El formato *JSON* es un formato de texto utilizado en el traspaso e intercambio de datos, utilizado en multitud de áreas debido a su sencillez. Consiste en colecciones clave-valor, donde los valores pueden ser numéricos, cadenas, booleanos, otras colecciones o arrays de cualquiera de estos tipos.

El motor de bases de datos *PostgreSQL* posee dos tipos de datos que permiten almacenar documentos *JSON* de forma nativa: **JSON** y **JSONB**. El primero almacena el documento como una cadena de texto y, cada vez que es necesario acceder o modificar una de sus claves, necesita *serializar* la cadena y convertirla a formato *JSON*.

El segundo tipo de datos, **JSONB**, es muy reciente: fue incorporado en la versión 9.5 de *PostgreSQL*. Internamente almacena el objeto en formato binario, lo que evita el proceso de *serialización*, además de contar con ciertas características adicionales, como la creación de índices para una clave del documento. Debido a sus nuevas capacidades y al aumento de velocidad que ofrece, se utilizará este último formato.

6.3. Estructura del sistema de búsqueda

El desarrollo de un sistema de búsqueda eficiente plantea grandes dificultades. Es una tarea en apariencia sencilla, pero gran parte de implementaciones fallan a la hora de devolver al usuario los resultados esperados. Para desarrollar un algoritmo que resulte útil hay que comenzar distinguiendo que tipos de búsquedas se van a realizar.vPor ejemplo, en el caso de una aplicación de lectura de libros, es posible que los usuarios quieran buscar desde un título o un autor hasta un extracto de un libro. Sin embargo, en un sistema como éste, donde predominan datos muy diferentes y de corta longitud, las búsquedas pueden ir desde valores numéricos (identificadores o números de serie)

hasta conceptos individuales muy variables: tres búsquedas diferentes, *fibra óptica*, *óptica fibra* y *fibras ópticas*, deberían retornar el mismo conjunto de resultados.

La técnica de búsqueda **full-text** es fundamental para cumplir con este último requisito, y su soporte nativo en **PostgreSQL** es uno de los principales motivos de la elección de dicho motor. Esta técnica analiza de forma individual cada una de las palabras de los campos buscados intentando encontrar los criterios de búsqueda solicitados y clasificándolos por su relevancia.

También puede encontrar términos relacionados al buscado gracias a una técnica llamada **stemming**, que clasifica los términos basándose en su raíz. Por ejemplo, los términos *búsqueda* y *buscador* comparten la misma raíz. Al buscar coincidencias con el criterio *buscar*, ambas serían seleccionadas como candidatas relevantes.

En la aplicación coexistirán dos sistemas de búsqueda diferentes, el sistema simple y el sistema avanzado. Es interesante comprobar que a nivel interno, como se explicará a continuación, el sistema de búsqueda simple es el más complejo de ambos.

En la interfaz de búsqueda simple, al usuario se le presenta únicamente un campo de texto que puede rellenar con la información que desee. Dependiendo de los datos que introduzca, el sistema será capaz de encontrar coincidencias en los siguientes campos:

- Identificador de una entrada, siempre que se haya buscado una cadena numérica.
- Número de serie de un elemento, en caso de buscar un único término cuya longitud sea mayor o igual de tres caracteres.
- Coincidencias *full-text* en la combinación de nombre, descripción y comentarios.

Por su parte, el panel de búsqueda avanzada permite al usuario filtrar los resultados por los campos que el desee, sin hacer ningún tipo de suposición. En la interfaz se muestran todos los datos que puede tener un elemento y se permite seleccionar cuales de ellos se utilizarán en el filtrado y el valor a buscar. En este tipo de búsqueda, los campos nombre, descripción y comentario también utilizan un tipo de búsqueda *full-text* pero, al contrario que en la búsqueda simple, no se realiza una comprobación de los criterios de búsqueda de forma simultánea en los tres.

Al no realizar una comprobación simultánea, una búsqueda como *fibra óptica*, el algoritmo simple será capaz de encontrar un elemento cuyo nombre contenga el término *fibra*, mientras que el término *óptica* sólo esté presente en la descripción.

El diagrama de flujo del algoritmo de búsqueda simple es el siguiente:

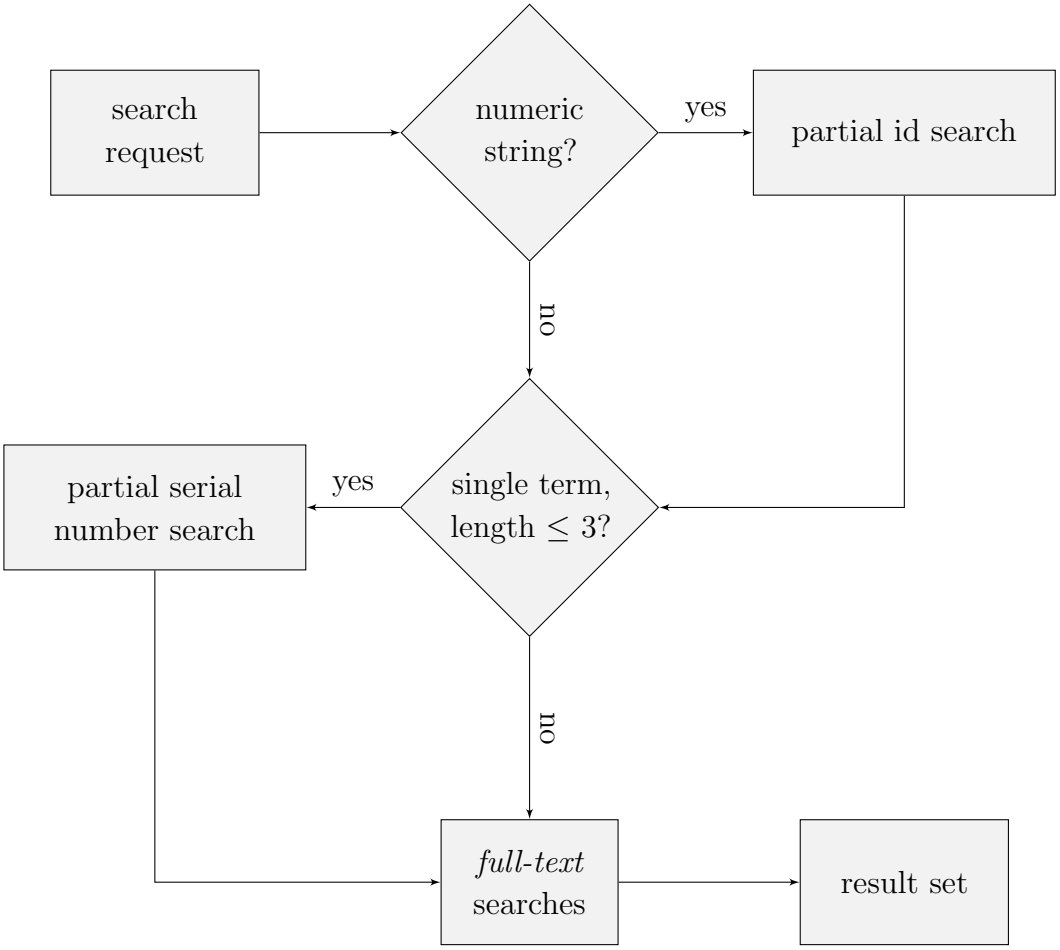


Figura 6.3: Diagrama de flujo del sistema de búsqueda simple

"The ultimate search engine would basically understand everything in the world, and it would always give you the right thing. And we're a long, long way from that."

— Larry Page, *cofundador de Google*

6.4. Estructura del front-end

Para la interfaz gráfica se ha optado por utilizar un *framework* muy popular en la actualidad, **Bootstrap**, desarrollado íntegramente por el equipo de desarrolladores de *Twitter*. Este framework incluye todas las estructuras para desarrollar una interfaz de forma sencilla, sin tener que dedicar esfuerzo en crear el *layout* y los estilos básicos.

Una de las ventajas de utilizar un *framework* tan extendido es la existencia de multitud de **temas**. Así, en lugar de invertir esfuerzo en crear un diseño, se utilizará el *template* **Gentelella**, que cubre las necesidades visualess de la aplicación.

Para reducir el consumo de recursos del sistema en los equipos clientes, se ha optado por incluir la menor cantidad posible de código *JavaScript*, limitando su uso a pequeños

aspectos visuales, y en mayor medida, a ciertas acciones del panel de administración. Así mismo, se ha elegido utilizar únicamente *JavaScript* puro, sin utilizar un *framework* como **jQuery** [30].

Esto supone un problema, ya que *Bootstrap* utiliza este *framework* para ciertos aspectos visuales integrados por defecto. Por tanto, ha sido necesario programar de forma nativa ciertas herramientas utilizadas a lo largo de toda la interfaz, como los diálogos modales en forma de *popup*, las notificaciones y los desplegables.

En los *scripts JavaScript* se ha optado por utilizar una metodología de programación mayoritariamente **funcional**. Aunque *JavaScript* no es un lenguaje funcional y, por tanto, no posee todas las herramientas necesarias para desarrollar utilizando este paradigma, el estilo de código utilizado sigue muchos de sus conceptos:

- Ausencia de estado. En la programación tradicional la evolución de un algoritmo se basa en el cambio de estado de variables *mutables*. Sin embargo, la programación funcional enfatiza la inmutabilidad de los datos, no permitiendo que una función produzca efectos secundarios sobre ninguna variable.
- Funciones de primer orden. *JavaScript* incluye esta característica, por la que una función es tratada como un objeto, pudiendo ser asignada a una variable, pasada como argumento a otras funciones o incluso retornada por una función.
- Técnica *map-reduce*. Consiste en un método de tratamiento de datos dividido en dos funciones, `map` y `reduce`. La primera función aplica el mismo algoritmo a cada elemento de un conjunto de datos. La segunda actúa sobre un conjunto y aplica una función que recombina sus elementos hasta devolver un único valor.

Para entender mejor esta metodología de programación, se muestra un simple ejemplo en *Python* que devuelve la suma de los cuadrados de un *array* numérico, utilizando funciones como parámetros, variables inmutables y las funciones `map` y `reduce`:

```
>>> from functools import reduce
>>> from operator import add, pow
>>> square = lambda x: pow(x, 2)
>>> numbers = [1, 2, 3, 4]
>>> squares = list(map(square, numbers))
>>> squaresum = reduce(add, squares)
>>> print(squaresum)
30
```

Figura 6.4: Ejemplo de paradigma funcional de programación

7. Seguridad

Ningún sistema es completamente seguro. Sin embargo, es importante tomar todas las medidas de seguridad posibles a la hora de desarrollar cualquier aplicación, para evitar posibles pérdidas de información o accesos no autorizados por parte de atacantes externos. A continuación se explican con detalle las medidas más importantes que se implementarán durante el desarrollo del sistema.

7.1. Password hardening

Los datos más críticos de cualquier aplicación son las contraseñas de sus usuarios. Obtener las credenciales de acceso de un usuario con permisos elevados permite a un atacante modificar o borrar la información almacenada en una aplicación, ocultando su identidad y suplantando la de otro individuo.

Un grave error es almacenar las contraseñas de los usuarios en texto plano. Si por un fallo de la aplicación las contraseñas quedasen expuestas, no se requeriría ningún esfuerzo adicional para acceder al sistema completo. Por ello, es imperativo codificarlas de alguna forma.

Comúnmente se habla de algoritmos de *encriptación* o de *cifrado* cuando se trata este tema. Sin embargo, es un concepto erróneo. Simplificando, una función de encriptación recibe un valor cualquiera y lo codifica utilizando una clave. Posteriormente, es posible decodificar el resultado obtenido utilizando una segunda clave, que puede ser o no la misma utilizada en la codificación (diferenciando así entre algoritmos simétricos y asimétricos).

$$c(message, key_{encrypt}) = cypher_{message}$$
$$c'(cypher_{message}, key_{decrypt}) = message$$

Las cadenas codificadas pueden ser decodificadas conociendo la clave de desencriptado, por lo que, aunque la dificultad para obtener acceso a las contraseñas originales aumenta, conseguir dicha clave supone un punto de fallo en el sistema.

La forma correcta de proteger las contraseñas de los usuarios es la utilización de algoritmos de *hasheo* o *digest*. A diferencia de los algoritmos descritos anteriormente, estas funciones mapean una valor de entrada de longitud indeterminada a una cadena de longitud finita, la cual no puede ser revertida al valor original.

$$h(message) = hash_{message}$$

Para una misma entrada, la salida resultante siempre será la misma. Por ello, a pesar de no existir una función inversa a la utilizada para hashear la entrada, es posible conocer el valor original utilizando las llamadas *rainbow tables*, tablas con un gran número de hashes precalculados. Para evitarlo, se utilizan utilizan *salts*, cadenas

aleatorias añadidas al mensaje original, y generadas de forma única para cada hash utilizando. De esta forma, dos datos de entrada iguales nunca generarán el mismo resultado.

$$\begin{aligned} r() &= salt_{random} \\ s(message, salt) &= concat(salt, message) = message_{salted} \\ h(s(message, r())) &= hash_{message_{salted}} \end{aligned}$$

Sin embargo, las funciones de hasheo están diseñadas para ser computacionalmente eficientes. Así, aún con todas las medidas utilizadas, es relativamente sencillo descubrir la contraseña utilizada usando ataques de fuerza bruta. Para evitarlo, se utilizan algoritmos de derivación de claves. Estas funciones generan un *salt*, lo añaden al mensaje original y posteriormente utilizan una función de hasheo para obtener una clave derivada. Posteriormente añaden el *salt* generado a la clave derivada y repiten la operación de *hasheo* con el nuevo mensaje generado un determinado número de veces, hasta obtener la clave final. De este modo, se consigue reducir la velocidad del ataque de fuerza bruta en un orden de magnitud relativo al número de iteraciones utilizado.

$$d(message, salt, iters) = \begin{cases} d(h(s(message, salt)), salt, iters - 1), & \text{if } iters > 1 \\ h(s(message, salt)), & \text{if } iters = 1 \end{cases}$$

En el apéndice **D** (**Benchmark de funciones de derivación**) se incluye un *script* para testear la velocidad de derivación de contraseñas en un sistema, utilizando para ello diferentes algoritmos de hasheo, números de iteraciones y longitudes de *salt*.

Por supuesto, estas medidas únicamente dificultan y encarecen computacionalmente el *crackeo* de contraseñas, no lo evitan por completo. Por ello, es fundamental instruir a los usuarios de la aplicación en el uso de contraseñas seguras, de alta longitud y con conjuntos de caracteres lo más amplios posibles (caracteres alfanuméricos y símbolos).

> *"Companies spend millions of dollars on firewalls, encryption and secure access devices, and it's money wasted, because none of these measures address the weakest link in the security chain."*
>
> — Kevin Mitnick (*The Condor*), *hacker estadounidense*

7.2. SQL injection

Las **inyecciones SQL** son ataques muy conocidos, producidos cuando la entrada de datos del usuario no es validada correctamente antes de ser utilizada en consultas a la base de datos.

A continuación se muestra en pseudocódigo un ejemplo sencillo de cómo con una consulta sencilla, rellenada con datos introducidos por el usuario, es posible eliminar todas las filas de una tabla de la base de datos:

```
input = "'; DELETE FROM entries WHERE 1 = 1 OR name = '"
query = "SELECT * FROM entries WHERE name = '" + input + "';"
execute(query)
SELECT * FROM entries WHERE name = '';
DELETE FROM entries WHERE 1 = 1 OR name = '';
```

Figura 7.1: Ejemplo de inyección *SQL*

Para evitarlo se utilizan consultas parametrizadas y, previamente a la ejecución de la consulta, se escapan todos los caracteres reservados en el motor de bases de datos utilizado. Así, aunque el usuario introduzca caracteres como comillas en la consulta, éstas no son interpretadas como un final de cadena:

```
input = "'; DELETE FROM entries WHERE 1 = 1 OR name = '"
query = "SELECT * FROM entries WHERE name = @name;"
execute(query, @name = escape(input))
SELECT * FROM entries
WHERE name = '\'; DELETE FROM entries WHERE 1 = 1 OR name = \'';
```

Figura 7.2: Código *SQL* saneado correctamente evitando una inyección

El paquete **SQLaAlchemy** utilizado en la abstracción de la capa de la base de datos aplica automáticamente las funciones de parametrizado y escapado de datos, evitando posibles *bugs* en cualquier consulta ejecutada.

7.3. Cross-site scripting

Otro de los ataques más comunes en aplicaciones web son los conocidos como **XSS**. Son similares a inyecciones SQL, pero en lugar de inyectar código en una base de datos, lo hacen en el momento en que la vista es generada, modificando la interfaz que verá el usuario final.

En el siguiente ejemplo en *HTML*, un código que originalmente muestra una imagen por pantalla, en realidad crea un mensaje de alerta al usuario con el texto `XSS!`:

```
input = '"><script>alert("XSS!");</script>'
htmltag = '<img src="' + input + '">'
print(htmltag)
<img src=""><script>alert("XSS!");</script>">
```

Figura 7.3: Ejemplo de inyección de código JavaScript

Escapando la entrada de datos, de forma similar a cómo se hace en *SQL*, todo carácter que *HTML* pueda interpretar de forma incorrecta es sustituido por su correspondiente referencia (secuencias de caracteres que representan caracteres concretos):

```
input = '"><script>alert("XSS!");</script>'
htmltag = '<img src="' + escape(input) + '">'
print(htmltag)
<img src="&quot;&gt;&lt;script&gt;alert(&quot;XSS!&quot;);&lt;/script&gt;">
```

Figura 7.4: *Input* de usuario saneada para evitar un ataque *XSS*

La librería de *templates* **Jinja2**, incluida en el *framework* **Flask**, dispone de un filtro de escapado de cadenas habilitado por defecto, por lo que todo valor almacenado en una variable es automáticamente saneado.

7.4. CSRF tokens

El último tipo de ataque contra el que se han tomado medidas es el llamado **Cross-Site Request Forgery**. Este tipo de ataque explota la confianza del servidor hacia un usuario, a diferencia de los ataques *XSS*, que se centran en la confianza del usuario hacia la aplicación.

Cuando un usuario con permisos de escritura quiere realizar una acción como puede ser borrar un elemento, normalmente puede hacerlo pulsando sobre un botón de la interfaz que realizará una petición a una ruta encargada de eliminar dicha entrada. Si un atacante es capaz de engañar al usuario, dirigiéndolo a esa ruta por otros medios, el usuario producirá la petición que borra una entrada, sin ser consciente de ello.

Dentro del protocolo **HTTP** existen multitud de métodos, o *verbos*, para realizar una petición, siendo los más comunes **GET** y **POST**. Las peticiones de tipo *GET* llevan en la propia dirección de la ruta todos sus parámetros necesarios, por lo que un ataque de este tipo es trivial. Por tanto, la primera acción a tomar es hacer que toda petición susceptible de modificar o eliminar cualquier tipo de información se realicen como peticiones *POST*, que requieren realizar la petición a través de un formulario web o de código *JavaScript*, dificultando la facilidad de explotar este fallo.

A pesar de utilizar peticiones *POST*, un atacante con la suficiente pericia puede llegar a inyectar un formulario una petición **AJAX**[10] y conseguir desencadenar una solicitud maliciosa. Para evitar que se puedan llegar a producir estas peticiones, en todas las acciones de escritura y administración se han implementado *tokens CSRF*.

Cada vez que el usuario solicita un formulario para modificar cualquier dato, el servidor genera un *token* único y aleatorio y lo asocia a la sesión. Para que la modificación pueda ser efectiva, el cliente ha de devolver dicho *token* al servidor. Como el *token* generado es aleatorio y único, no es posible de forma trivial para el atacante obtener un *token* válido para crear una petición maliciosa correcta.

[10] *Asynchronous JavaScript And XML*: tecnología existente en *JavaScript* para realizar peticiones web asíncronas sin necesidad de recargar la página.

7.5. HTTPS

Se ha optado por no utilizar el protocolo *HTTPS* (**HyperText Transfer Protocol Secure**) a la hora de acceder a la aplicación. En principio, esto parece incoherente después de haber aplicado las medidas de seguridad anteriores. Sin embargo, se ha considerado no ser necesario por varios motivos.

Al transferir la información de forma encriptada con claves asimétricas, se asegura que, en caso de que un atacante capaz de interceptar la comunicación (por ejemplo, a través de un ataque **MITM**[11]), ésta no sea legible. Sin embargo, al ser ésta una aplicación destinada a utilizarse dentro de la red interna de la Universidad, controlada por el equipo informático de la misma y sin posibilidad de acceso interno, es evidente que estos tipos de ataques no pueden ocurrir.

Basándose en esta última afirmación, se podría argumentar que el resto de medidas de seguridad implementadas también son innecesarias. Sin embargo, siempre es posible que un virus o un atacante consiga hacerse con el control de una máquina local, pudiendo llegar a explotar las vulnerabilidades antes descritas. En caso de que esto ocurriera, la encriptación añadida por el protocolo *HTTPS* sería completamente ineficaz, ya que esta medida de seguridad se enfoca únicamente en la comunicación entre el cliente y el servidor.

Por otro lado, el protocolo *HTTPS* depende de la existencia de un par de claves de confianza utilizadas para encriptar y desencriptar las transferencias. Éstas claves son emitidas por autoridades certificadores de confianza y suelen ser de pago, con un coste bastante elevado. Recientemente ha aparecido una alternativa gratuita, **Let's Encrypt** [34], pero depende de que la aplicación sea accesible desde una red externa. Es posible crear un par de claves autofirmadas de forma manual y obligar a cada uno de los sistemas cliente a confiar en ellas. Sin embargo, no proporcionan el mismo nivel de seguridad y únicamente crean una falsa sensación de seguridad al usuario.

En caso de querer utilizar la aplicación en un equipo en el cual no se haya forzado la confianza en el certificado autofirmado, el navegador mostraría por pantalla un mensaje de error que el usuario ha de aceptar para poder acceder a la aplicación. En este momento, un atacante que intercepte las comunicaciones podría sustituir el certificado utilizado por otro diferente. El mensaje de error también aparecería en este caso, pero el usuario seguiría aceptando la navegación al no ser consciente del ataque.

Por tanto, se ha mantenido el uso del protocolo no seguro *HTTP*, ya que el entorno en el que se accederá a la aplicación no requiere de esta medida de seguridad, la cual únicamente dificultaría la navegación del usuario al requerir acciones por su parte en su equipo para configurar la confianza del navegador en el certificado autofirmado.

[11] *Man In The Middle*: un ataque en el que es posible interceptar e incluso modificar la información de una transferencia sin que quede constancia de ello.

8. Desarrollo

El desarrollo de la aplicación se dividirá en diferentes fases independientes, siguiendo un orden de prioridades y de dependencias que, a su vez, sirven como *checkpoints* para medir la evolución del proyecto y el cumplimiento de los plazos marcados.

8.1. Instalación

El proceso de instalación del sistema comienza, lógicamente, por la instalación del sistema operativo **CentOS 7**. En este caso, este proceso no es un paso necesario, ya que el servicio informático de la Universidad proporciona para este proyecto una instancia de tipo **KVM**[12] [1] en sus servidores de virtualización.

Sin embargo, durante el desarrollo del proyecto se ha realizado una instalación limpia del sistema operativo en una máquina independiente. El proceso de instalación en esta máquina, aunque muy sencillo, se encuentra explicado por completo en el apéndice **E** (**Instalación del sistema operativo CentOS 7**).

Una vez instalado el sistema operativo, se ejecutará el *script* de instalación desarrollado, el cual se muestra en el apéndice **F.1** (**Script de instalación**). Este *script* se ha ideado para que el proceso de instalación en cualquier escenario sea automático y no requiera de ninguna interacción con el usuario. Las tareas que realiza son las siguientes:

- Actualización de todos los paquetes del sistema e instalación de las herramientas necesarias para la ejecución de la aplicación: el lenguaje de programación *Python*, el gestor de bases de datos *PostgreSQL* y el servidor web *Nginx*.
- Creación automática de los usuarios de sistema necesarios, y configuración de los parámetros de todas las herramientas utilizadas.
- Creación del entorno virtual de ejecución de la aplicación e instalación de todos los paquetes *Python* necesarios para su funcionamiento: el *framework Flask* y sus *plugins*, el *software* necesario para la comunicación *wsgi* y el *driver* de conexión a la base de datos.
- Descompresión de la aplicación en los directorios especificados en apartados posteriores y ejecución inmediata de la misma.
- Configuración de todas las políticas de seguridad del módulo *SELinux*[13] [33] y del *firewall* utilizado, de tal manera que sólo se encuentren accesibles los *endpoints* disponibles para acceder a la aplicación.

[12] método de virtualización disponible en sistemas *Linux*, el cual permite hacer uso de las opciones de virtualización nativas del *hardware*, sin necesidad de realizar modificaciones en el *kernel anfitrión*.

[13] módulo de seguridad disponible en el *kernel* de muchas distribuciones *Linux*, que proporciona mecanismos y políticas de seguridad para interceptar o bloquear cualquier acción a realizar en el sistema.

8.2. Estructura de archivos

Toda los ficheros de la aplicación se alojarán dentro del directorio `/var/www/photonics/`. En la instalación se ha creado un entorno virtual de Python dentro de ese directorio, y se han instalado todos los paquetes necesarios para el funcionamiento de la aplicación. Para evitar que todos los ficheros relacionados con este entorno virtual sean accesibles desde el servidor web, se creará un subdirectorio `htdocs` que será la raíz accesible desde el servidor. Además, se crearán dos subdirectorios, `files` e `images` para almacenar los ficheros estáticos relativos a los elementos del inventario.

El directorio principal contiene también un fichero, `uwsgi.ini`, que contiene todos los parámetros de configuración necesarios para arrancar el servicio de comunicación entre el servidor web y los scripts en *Python*.

Por otro lado, existe también un fichero `babel.cfg` que se utilizará a la hora de crear los ficheros de traducción de la aplicación a diferentes idiomas.

La raíz de la aplicación, `htdocs/`, se estructura en los siguientes subdirectorios:

- `models`: contiene los modelos utilizados por la aplicación.
- `views`: contiene los controladores y rutas de la aplicación.
- `templates`: contiene las vistas y macros a renderizar.
- `forms`: contiene las clases de todos los formularios utilizados.
- `static`: contiene ficheros estáticos (estilos, imágenes, tipografías y *scripts*).

En la raíz de la aplicación hay, además, varios scripts *Python* fundamentales para el correcto funcionamiento del sistema:

- `config.py`: contiene todos los parámetros de configuración necesarios para el correcto funcionamiento de la aplicación (cadenas de conexión, idiomas soportados y listados de directorios).
- `app.py`: es el script principal desde el que se ejecuta la aplicación, tanto en modo *debug* como en producción.
- `shared.py`: contiene el objeto de conexión a la base de datos. Este script no tiene otra utilidad, pero se ha creado para poder conectar a la base de datos sin crear referencias circulares con otros scripts.
- `utils.py`: listado de funciones y utilidades básicas para realizar ciertas acciones comunes a toda la aplicación.
- `photon.py`: script con todas las funciones relacionadas con el *framework* **Flask** y todos sus plugins. Es la base de la aplicación.

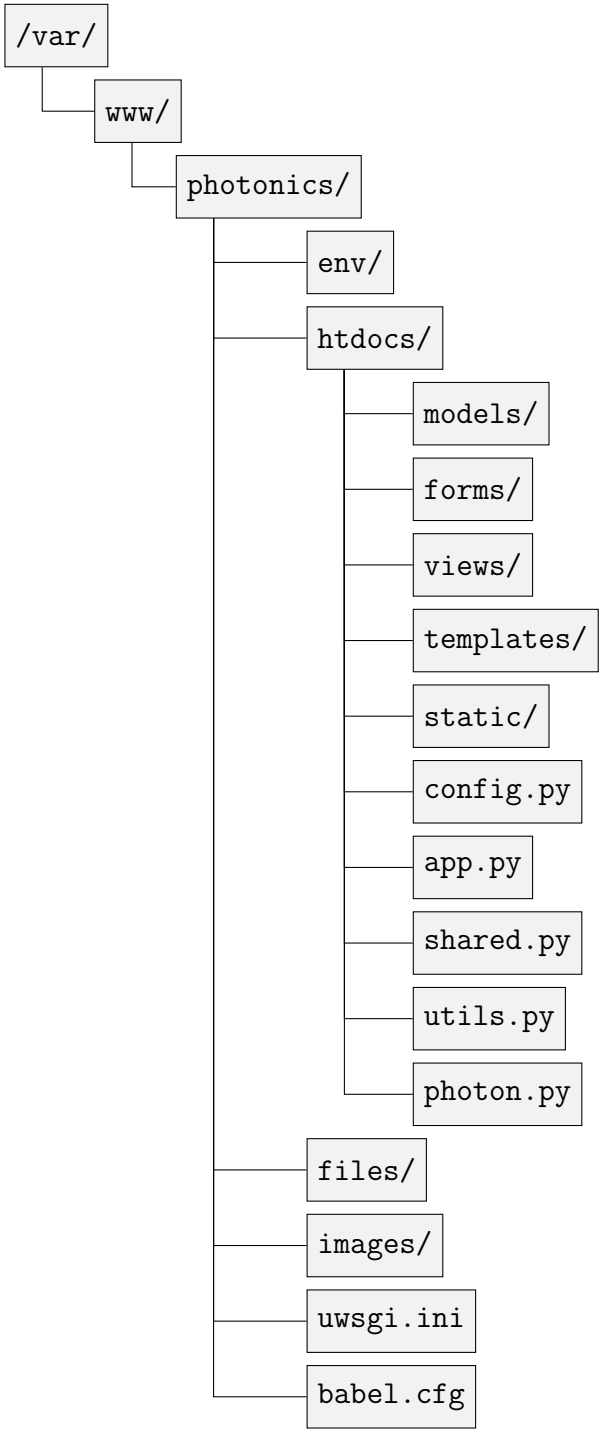


Figura 8.1: Árbol de ficheros de la aplicación

8.2.1. uwsgi.ini

Como ya se ha explicado anteriormente, para poder servir la aplicación y recibir peticiones por los usuarios, es necesario comunicar el servidor web **nginx** con la aplicación, a través de un protocolo llamado **wsgi**. A la hora de arrancar el servicio de comunicación es necesario configurar varios parámetros, como cual es el script a

ejecutar o el número de procesos que ha de crear. Todos estos parámetros se almacenan en este fichero que se encuentra en la raíz del servidor.

El fichero completo se incluye en el apéndice **G.1** (**Fichero de configuración del protocolo *wsgi***). A continuación se explican con mayor detalle los parámetros necesarios para ejecutar la aplicación correctamente:

- Directorio principal de la aplicación y *script* a ejecutar. Es necesario indicar cual es es punto de entrada a la aplicación que recibirá las peticiones. Para ello se utilizan tres parámetros:
  `base=/var/www/photonics/htdocs/` indica el directorio raíz de la aplicación
  `module=app` especifica el *script Python* a ejecutar
  `callable=application` determina la variable que contiene el contexto *Flask*

- *Path* del entorno virtual. Ya que la aplicación no se ejecuta en el entorno global del sistema, sino que se ejecuta dentro de un entorno virtual, es necesario establecer los datos del entorno:
  `home=%(base)../env/` dirección raíz del entorno virtual
  `pythonpath=%(base)` y `chdir=%(base)` path donde se ejecutará la aplicación

- Credenciales de sistema con las que se ejecutará la aplicación. Se indican el usuario y grupo creados se han creado durante la instalación:
  `uid=photonics` nombre del usuario
  `gid=nginx` nombre del grupo

- Nombre y permisos del socket *socket* a crear. La comunicación entre el servidor web y el proceso *Python* se realiza a través de un *socket Unix* que se creará en el directorio de la aplicación:
  `socket=/var/www/photonics/%n.sock` ubicación del fichero a crear
  `chown-socket=photonics:nginx` asignación del *socket* al usuario de la aplicación
  `chmod-socket=660` permisos de sistema del *socket*

8.2.2. photon.py

El *framework* **Flask** incluye multitud de funciones que facilitan el desarrollo web, evitando escribir grandes cantidades de código repetitivo para simples funciones, básicas en cualquier aplicación. Sin embargo, para las necesidades de este entorno, es necesario crear otra capa de abstracción más, que ahorrará mucho trabajo posterior. En este fichero se almacenarán todas las funciones de ayuda para el desarrollo de la aplicación. En el apéndice **G.2** (**Abstracción sobre *Flask***) se incluye el *script* completo. Se listan a continuación las funciones más importantes incluidas en este fichero:

- Inicialización de la aplicación y todos sus plugins asociados: a la hora de crear la aplicación es necesario configurar los parámetros y opciones incluidas en el

fichero **config.py**. Además, la mayoría de plugins utilizados requieren asociarse al contexto de la aplicación para funcionar.

- Selección automática de idioma actual: el idioma utilizado para renderizar cada una de las vistas depende del usuario activo que realice la petición.
- Configuración de rutas y nombrado automático de las mismas: por defecto, *Flask* incluye un método para asociar cada controlador a una ruta concreto, utilizando el *decorator* `@flask.route('/route/path')`. Para simplificar la programación de nuevos controladores, se ha desarrollado un método que transforma automáticamente el nombre del controlador a una ruta válida.
- Gestión de credenciales de autenticación y permisos de acceso: probablemente el área más importante de este fichero. Se automatiza el proceso de comprobación de acceso a cada controlador, comprobando automáticamente si una petición procede de un usuario *logueado* con permisos suficientes para acceder a la ruta solicitada.
- Renderizado automático de *templates*: el *layout* básico de la aplicación requiere de ciertos datos provenientes de varios modelos, que han de mostrarse siempre, sin importar que controlador se haya solicitado. Para evitar enviar esta información en cada uno de los controladores, se simplifica el proceso, inyectando los datos necesarios al terminar de procesar el controlador.

8.3. Rutas y restricciones

La principal característica de *Flask* es su capacidad de *mapear* una función a una dirección *URL* concreta. Este método de enrutado de *URL* a controladores permite generar un tipo de direcciones llamadas **semantic URL**[14]. Es habitual ver aplicaciones web con un formato de direcciones poco legibles para el usuario y complicadas de recordar, como `http://host/path?parameter1=value&parameter2=value`. Al utilizar *URL semánticas* es posible generar direcciones mucho más visuales, que permiten reconocer fácilmente el recurso que será ofrecido por el servidor.

La aplicación se estructurará en tres tipos de rutas diferentes, aquellas relacionadas con el inventario, las utilizadas para gestionar el panel de administración y todas las pertenecientes al sistema de usuarios. Existirán tres ficheros en el área de controladores, uno para cada tipo de ruta:

- **Rutas del sistema de inventario**: el *path* de todas ellas comenzará siempre por `/inventory`. El listado de elementos y categorías, las herramientas de búsqueda y los sistemas de inserción y modificación de elementos se gestionarán desde este tipo de rutas, asociadas todas ellas al fichero `inventory.py`.

[14] recursos con direcciones legibles para el usuario, en lugar de las tradicionales URL extensas en las que se muestra el listado de parámetros enviados al servidor.

- **Rutas del sistema de administración**: el panel principal de administración y todas sus opciones y herramientas se controlarán desde las rutas con path `/admin` y alojadas en el fichero `admin.py`.
- **Rutas del sistema de usuarios**: se encuentran en el fichero `account.py`. La más importante de ellas es la de autenticación de usuarios a la hora de entrar al sistema. También permiten acceder al perfil de cada usuario, modificar sus datos y cambiar o reestablecer su contraseña.

A la hora de controlar los permisos de acceso se creará un *decorator*[15] en el fichero `photon.py` que se encargará de inyectar en cada controlador el código necesario para comprobar la correcta autenticación de un usuario y si posee los permisos de acceso necesarios para acceder a dicho controlador.

```
def authenticated(anonymous=False, write=None, admin=None):
  def decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
      if anonymous and current_user.is_authenticated:
        return redirect('/')
      elif not anonymous and not _app.config['LOGIN_DISABLED']:
        if not current_user.is_authenticated:
          return redirect(_login_manager.login_view)
        elif not current_user.can_login:
          logout_user()
          return redirect(_login_manager.login_view)
        elif not current_user.can_admin:
          if admin or write and not current_user.can_write:
            return redirect('/')
      return func(*args, **kwargs)
    return functools.update_wrapper(wrapper, func)
  return decorator
```

Figura 8.2: *Decorator* para gestionar los permisos de acceso de un controlador

8.4. Base de datos

Una de las múltiples ventajas de utilizar una librería como **SqlAlchemy** es la posibilidad que ofrece de abstrarse por completo del motor de bases de datos. Gracias al uso de la técnica **ORM** (*Object-Relational Mapping*), que relaciona los objetos creados en *Python* a tablas y columnas creadas de forma automática, no es necesario utilizar la interfaz de gestión de la base de datos para crear la estructura de tablas.

[15]patrón de diseño muy utilizado en *Python* que permite modificar de forma dinámica la funcionalidad de una una función sin necesidad de modificar directamente su código.

Al definir los modelos en la propia aplicación, la librería crea automáticamente las tablas, atributos, índices y relaciones necesarias. Así a la hora de modificar un modelo, no es necesario realizar la refactorización en dos capas diferentes del mismo, ya que cualquier cambio en los modelos se aplica automáticamente a la base de datos.

Por norma general, esta metodología de desarrollo posee otra ventaja: poder sustituir de forma totalmente transparente el motor de bases de datos por otro, sin que la funcionalidad de la aplicación se vea afectada. Sin embargo, en este caso, al utilizar ciertas características específicas de **PostgreSQL** (incluidas también de forma nativa en la librería), no es posible realizar esa migración de una forma tan sencilla.

Dentro del directorio *models* se almacenarán todos los modelos utilizados por la aplicación, así como todas las funciones y métodos que actúan sobre ellos directamente. La estructura básica de un modelo se especifica con detalle en el apéndice **G.3.1** (**Modelos**). En un modelo se pueden especificar directamente todas las claves, índices y referencias a otros modelos directamente, y gracias a que la librería *SqlAlchemy* ya provee de las herramientas de consulta, edición y modificación de datos en la base de datos, no es necesario realizar ningún desarrollo extra.

8.5. Interfaz gráfica

El primer paso a la hora de desarrollar la interfaz gráfica es la instalación del *framework* **Bootstrap** y el tema **Gentelella**. Gracias a estas dos utilidades, prácticamente la totalidad de estilos y estructuras utilizadas en el *front-end* se encuentran disponibles sin mayor esfuerzo.

Será necesario realizar ciertos cambios, así como añadir nuevas estructuras a las hojas de estilo *CSS* del *template* utilizado, para adecuar varios aspectos del diseño a la información mostrada por pantalla. También se realizarán modificaciones de cara a conseguir una interfaz *responsive* optimizada a todo tipo de pantalla. Estas modificaciones se reflejarán en el fichero `photon.css` del subdirectorio `static`.

Como se ha explicado anteriormente, se ha optado por no utilizar el *framework* **jQuery**, por lo que es necesario implementar ciertas funcionalidades visuales que ya integra *Bootstrap* por defecto. En el fichero `photon.js` del subdirectorio `static` se integrarán todas estas herramientas visuales: los *diálogos modales*, los *popups* de notificaciones, los *tooltips*, los menús *desplegables* en forma de árbol y las estructuras *colapsables*. Además, en este fichero también se incluirán herramientas visuales comunes a todas las áreas de la aplicación, como los sistemas de búsqueda, explicados posteriormente.

Teniendo en mente la legibilidad y la reusabilidad del código, a la hora de diseñar las vistas se va a hacer un uso muy frecuente de una característica disponible en el sistema de *templates* **Jinja2**: las **macros**. En esencia, una macro es similar a una función definida en cualquier lenguaje de programación, la cual puede recibir parámetros y retornar código *HTML* que se mostrará por pantalla. Así, estructuras que se reutilizarán en varias zonas de la aplicación, como pueden ser el desplegable

de categorías o el visualizador de entradas, quedan aisladas del *layout* de la vista, pudiendo reutilizarse y hasta modificar su aspecto de forma sencilla.

8.6. Rutas de inventario

El área de inventario es, sin duda, el núcleo de la aplicación. La mayor parte del tiempo transcurrido en la misma será en alguno de sus apartados, por lo que es, también, el área sobre el que se han centrado la mayor parte de los esfuerzos de desarrollo.

La primera funcionalidad que posee es la visualización de un conjunto de entradas de forma simultánea, mejorando así la antigua interfaz, en la que solamente se podía visualizar un único elemento a la vez. Este tipo de visual se utilizará a la hora de filtrar los elementos por categoría, así como en el sistema de búsqueda que se implementará posteriormente. En esta interfaz, los elementos se mostrarán como módulos desplegables, contraídos por defecto, pudiendo observar únicamente su identificador y su nombre. Al expandir uno de estos elementos se mostrarán los datos más importantes del mismo: categoría, estado, descripción, comentarios y, en caso de existir, la información de uso y el material digital asociado a la misma.

En la visual de entradas múltiples existirá, para cada una de ellas, un botón que permitirá mostrar toda la información de una única entrada. En esta visual aparecerán el resto de campos, además de una imagen, en caso de tener alguna asociada.

Finalmente, se desarrollará una interfaz para insertar nuevos elementos, así como modificar elementos existentes. En esta visual, que tomará la forma de un formulario distribuido de igual manera que la visual de un único elemento, además de poder modificar todos los campos asociados a la entrada, será posible añadir archivos e imágenes a la misma. Estos ficheros podrán subirse a la aplicación a través de las herramientas nativas ofrecidas por el navegador y, en caso de que éste permita acciones de *drag&drop*, podrán arrastrarse directamente a la ventana de la aplicación desde cualquier otro programa. Para ello, se ha desarrollado un *script* llamado `upload.js`, situado en el subdirectorio `static/`, encargado de gestionar ambas formas de subida.

```
var hasAdvancedUpload = function() {
  var hop = Object.prototype.hasOwnProperty;
  if (!hop.call(window, 'CustomEvent')) {
    return false;
  }
  if (hop.call(window, 'FormData') && hop.call(window, 'FileReader')) {
    var div = document.createElement('div');
    return 'draggable' in div || ('ondragstart' in div && 'ondrop' in div);
  }
  return false;
}();
```

Figura 8.3: *Script* para comprobar la funcionalidad *drag&drop* del navegador

8.7. Panel de administración

A diferencia del resto de zonas, el panel de administración sí que requiere de un mayor uso de *scripts JavaScript* para facilitar su uso. Exceptuando la zona de gestión de usuarios, la cual se renderizará en una vista aparte por su complejidad, el resto de áreas a gestionar desde el panel se realizarán desde la interfaz principal del mismo.

Se han diseñado las acciones que es posible realizar en el propio panel (como inserción, modificación o borrado de datos) de manera que su ejecución sea la misma, sin tener en cuenta el área sobre el que actúan. Esto permite generalizar el código del *script* del lado del cliente, abstrayéndolo del área sobre el que actúan. Esto facilita la inclusión de forma sencilla de nuevas funcionalidades o incluso el añadido de áreas nuevas al panel utilizando funcionalidades ya existentes.

```
;(function(window, document, undefined) {
  'use strict';
  var createEvents = (function(actions) {
    return function(arr) {
      (function(methods, selector, route, options) {
        methods.forEach(function(v) {
          var elements = document.querySelectorAll(selector + ' .' + v);
          Array.prototype.forEach.call(elements, function(w) {
            w.addEventListener(
              'click',
              actions[v](selector, route && (route + v), options),
              false);
          });
        });
      }).apply(null, arr);
    }
  })((function() {
    return { /* list of actions (add, edit...) */ }
  })());
  [ /* list of areas with actions  */ ].forEach(createEvents);
})(window, document);
```

Figura 8.4: Esquema del *script* de funciones del panel de administración

8.8. Sistema de búsqueda

Como se ha definido anteriormente, el sistema de búsqueda se dividirá en dos áreas diferentes, la búsqueda simple y la búsqueda avanzada. Las búsquedas se realizarán a través de formularios, por lo que, según las directrices de seguridad especificadas en el apartado **7.4** (**CSRF tokens**), las búsquedas deberían ser peticiones al servidor utilizando el método **POST**. Sin embargo, al tratarse de peticiones de sólo lectura, se

implementarán técnicas para poder ejecutar peticiones de búsqueda desde una *URL* con una petición **GET**.

8.8.1. Búsqueda simple

Este algoritmo requiere un parámetro con la cadena de texto a buscar, por lo que la ruta, de tipo *GET*, tendrá la forma `/inventory/search/<path:search_query>`. El tipo de parámetro `path` es similar a `string`, pero permite incluir **slashes** (caracteres `/`) en la cadena de texto.

La desventaja de los formularios *HTML* es que no es posible realizar peticiones *GET* utilizando el formato de *URL semánticas*. Por tanto, es necesario preparar un pequeño *script* en *JavaScript*, encargado de interceptar el envío del formulario de búsqueda y redireccionarlo a la ruta correspondiente.

```
var form = document.getElementById('search');
var input = form.querySelector('input');
function evt(e) {
  e && e.preventDefault() && e.stopPropagation();
  var search = encodeURIComponent(input.value.trim());
  search.length && (window.location.href = '/inventory/search/query/' + search);
}
form.addEventListener('submit', evt, false);
```

Figura 8.5: Inyección de ruta semántica en el envío del formulario de búsqueda

En el lado del servidor, es necesario implementar el diagrama de flujo definido en la figura **6.3**. Cada una de las búsquedas parciales se ejecutará en una consulta diferente a la base de datos, ya que la complejidad de realizar la búsqueda en una única consulta es muy elevada. Como las diferentes consultas pueden devolver resultados duplicados, es necesario filtrarlos utilizando colecciones de tipo **set**[16]. Sin embargo, *Python* no cuenta con un un tipo de datos ordenado y sin duplicidades, por lo que, para evitar perder el orden de la clasificación de prioridad de las búsquedas *full-text* es necesario instalar el paquete **OrderedSet** con la instrucción `pip install orderedset`.

Es necesario indicar al módulo *OrderedSet* la forma en la que ha de detectar duplicidades de objetos. Las clases y objetos predefinidos en el entorno de programación cuentan con un método `__hash__` que devuelve un valor único en cada instancia distinta de un objeto. Sin embargo, es necesario implementar este método en cualquier clase definida por el usuario, ya que este es el método que ejecutará el módulo para diferenciar objetos. En este caso, ya que los identificadores de los modelos siempre son únicos, es posible definir este método para que su valor de retorno sea dicho identificador.

[16] colecciones que no permiten la inserción de objetos duplicados, para lo cual comparan el elemento a insertar con un algoritmo definido por un algoritmo de *hash* definido en el objeto.

A la hora de realizar búsquedas *full-text*, es necesario crear un vector de búsqueda en cada columna sobre la que se desea buscar. Es posible crear temporalmente este vector en la misma consulta, pero esto supone un alto coste en tiempo. Por ello, es recomendable crear columnas nuevas en la propia base de datos y precalcular estos vectores en el momento de la inserción. El paquete **SqlAlchemy-Searchable** (instalable con el comando `pip install sqlalchemy-searchable`) permite incluir este tipo de dato en los propios modelos, permitiendo así trabajar con ellos directamente a través de la librería **SqlAlchemy**.

```
name_search_vector=db.Column(TSVectorType('name'))
description_search_vector=db.Column(TSVectorType('description'))
comments_search_vector=db.Column(TSVectorType('comments'))
full_search_vector=db.Column(TSVectorType('name','description','comments'))
```

Figura 8.6: Campos de tipo vector incluidos en el modelo **Entry**

Gracias a la existencia de estos campos, el tiempo invertido en realizar cualquier búsqueda se reduce, siendo ésta casi instantánea. Así pues, sólo queda implementar en la ruta de búsqueda el diagrama de flujo anterior, el cual se detalla en el apéndice **G.4.1** (**Búsqueda simple**).

8.8.2. Búsqueda avanzada

El algoritmo de búsqueda avanzada presenta problemas a la hora de generar una *URL* semántica, ya que la cantidad de campos es muy elevada, además de la existencia de campos dinámicos que no son parametrizables en forma de variables en la ruta de la petición. Por ello, se ha decidido tomar un camino alternativo, en el que se *serializan* todos los datos de búsqueda en un único parámetro. Así, es posible crear una ruta similar a la de la búsqueda simple, `/inventory/advanced_search/<string:search_query>`. En este caso también es necesario utilizar un parámetro de tipo `path`, ya que la búsqueda se codificará en formato **base64**, el cual utiliza un alfabeto de caracteres alfanúmericos con distinción entre mayúsculas y minúsculas, además de los caracteres `+`, `/` y `=`.

El protocolo *HTTP* no limita el tamaño en caracteres de una dirección *URL*, tal como se indica en el documento RFC[17] 2616 [35]. En el documento RFC 7230 [36] se recomienda evitar las direcciones con más de 8000 caracteres, sin embargo, existen navegadores con un límite máximo de 2000 caracteres.

Para evitar enviar una cadena de caracteres excesivamente larga, en lugar de inyectar una petición **GET** a la hora de enviar el formulario, se realizará una petición **POST** normal. Al recibirla, el servidor analizará el listado de parámetros, eliminando

[17] *Request for Comments*: publicaciones de la *IETF* (**Internet Engineering Task Force**) acerca de aspectos y buenas prácticas sobre el funcionamiento de Internet.

aquellos que estén vacíos o con caracteres sobrantes, como espacios iniciales o finales. Posteriormente, se *serializarán* los criterios de búsqueda y se codificarán en *base64*. Finalmente, se redirigirá al usuario a la ruta indicada previamente, la cual será la encargada de realizar la búsqueda. En el apéndice **G.4.2** (**Búsqueda avanzada**) se muestra todo el código que realiza esta tarea.

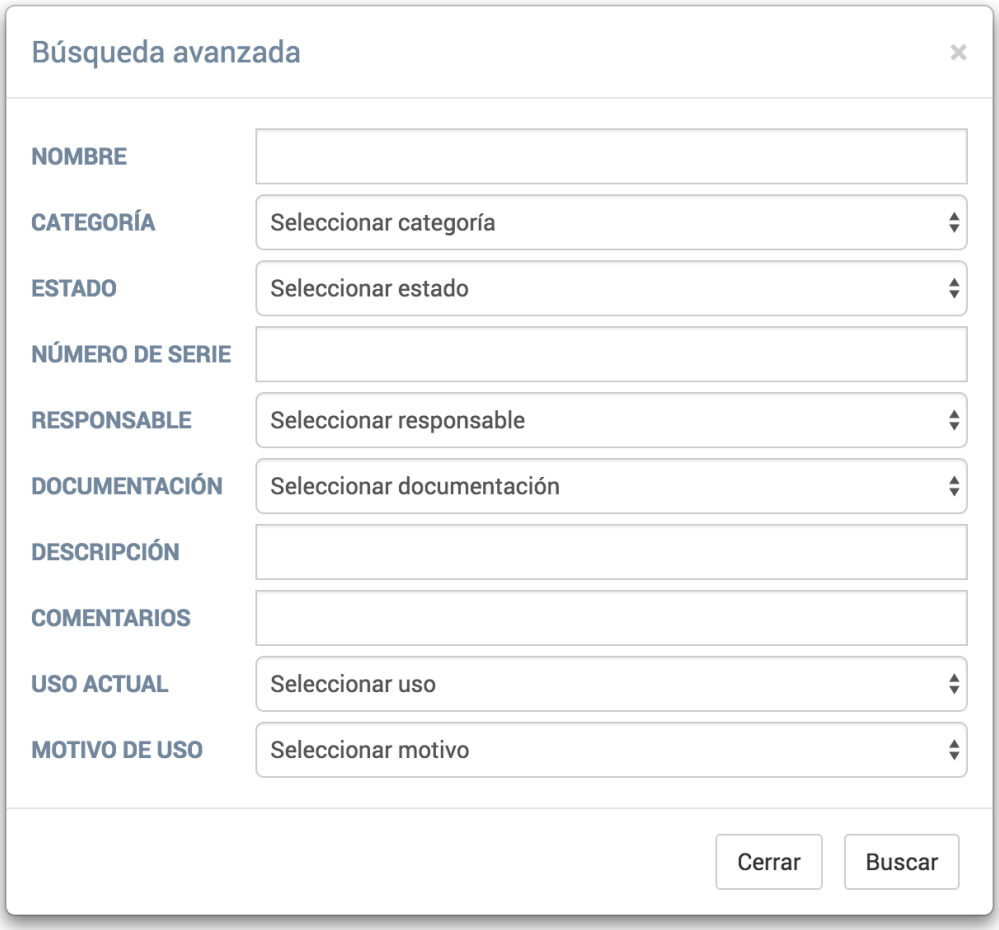


Figura 8.7: Diálogo modal de búsqueda avanzada

8.9. Actualizaciones

Antes de comenzar el desarrollo del proyecto se planteó la posibilidad de que las actualizaciones del sistema se descargasen e instalasen periódicamente de forma automática, para así mantener el servidor protegido contra posibles fallos de seguridad. Posteriormente se comprobó que estas actualizaciones automáticas pueden llegar a causar problemas o incompatibilidades en caso de realizar alguna modificación en áreas críticas del sistema como el *kernel*, por lo que esta idea quedó descartada. En su lugar, se optó por incluir un método de búsqueda e instalación de actualizaciones a través de la propia interfaz del panel de administración. Así, en caso de que el administrador decida realizar una actualización del sistema, puede hacerlo fácilmente.

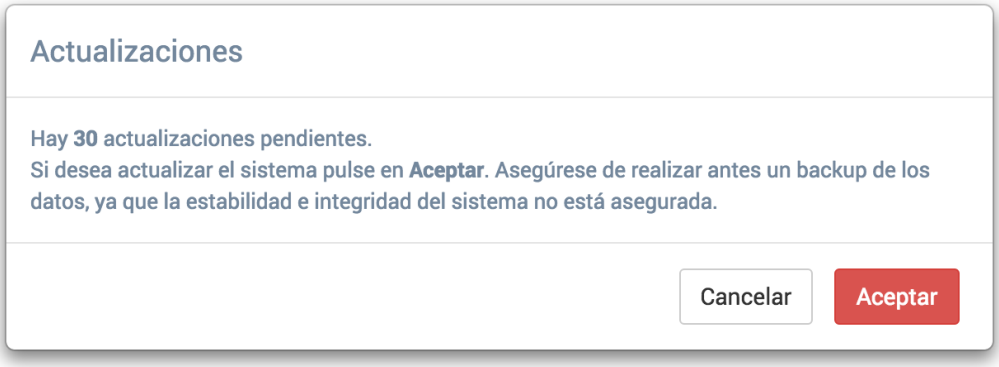


Figura 8.8: Interfaz de actualización en el panel de administración

Para el desarrollo de esta funcionalidad se ha aprovechado la capacidad de *Python* de ejecutar comandos del sistema a través del módulo `subprocess`. Es necesario ejecutar dos comandos diferentes para el funcionamiento de este sistema: uno para comprobar el número de actualizaciones y otro para realizar la instalación y reinicio del servidor.

```
command = "/bin/yum check-update --quiet"
result = subprocess.Popen(
  command.split()
  stdout=subprocess.PIPE,
  stderr=subprocess.PIPE)
updates = str(result.stdout)
updates = updates.split('\n')
updates = len([x for x in updates if re.match('^a-zA-Z0-9', x)])
```

Figura 8.9: *Script* de comprobación del número de actualizaciones pendientes

A través de la función `Popen` no es posible ejecutar varios comandos de forma simultanea. Para realizar la actualización del sistema se requieren dos comandos, el primero para realizar la actualización y el segundo para reiniciar el sistema. Para solucionarlo, se creará un archivo *shell script* que ejecutará ambos comandos, y desde la interfaz de *Python* se llamará a dicho *script*.

```
#!/bin/sh
/usr/bin/sudo /bin/yum -y update && /usr/bin/sudo /sbin/reboot
```

Figura 8.10: *Shell script* de actualización del sistema

En cualquier caso, es necesario tener en cuenta que cualquier actualización puede causar problemas en el correcto funcionamiento del sistema, siendo cualquier problema o error responsabilidad del administrador, por lo que es recomendable hacer una copia de seguridad de todos los datos, así como un *snapshot* de la máquina virtual antes de realizar esta actualización.

8.10. Copias de seguridad

Es necesario diferenciar los dos sistemas de copia de seguridad disponibles, el manual y el automático. Las copias de seguridad manuales puede efectuarlas un administrador en cualquier momento a través del portal de administración de la aplicación. A través de este método se realizarán de forma independiente los *backups* de la base de datos y de los ficheros almacenados.

Figura 8.11: Zona de *backup* del panel de administración

Por su parte, el sistema de copia de seguridad automática crea un fichero **zip** con los *backups* de los datos de la aplicación y de los ficheros almacenados. Los archivos contenidos en el fichero comprimido son compatibles con los generados manualmente, pudiendo restaurar de forma manual un *backup* generado a través de este sistema.

Para ejecutar esta copia de seguridad automática es necesario poseer una cuenta de usuario de la aplicación con credenciales de administración y realizar una petición a una ruta concreta, indicando las credenciales del usuario. Esta petición se puede automatizar en cualquier sistema, utilizando herramientas como `cron` en *Linux* para realizarla en intervalos de tiempo definidos.

`http://fotonica.unavarra.es/admin/backup/auto/<username>:<password>`

Importante: la restauración de *backups* no es *safe-thread*[18], por lo que la operación fallará si dos administradores intentar restaurar un *backup* simultáneamente.

8.11. Internacionalización

Toda la aplicación se ha diseñado teniendo en mente la posibilidad de adaptarse a múltiples idiomas. En lugar de *hardcodear*[19] cada una de las cadenas de texto visibles en la interfaz gráfica, se han utilizado *tokens* identificativos que definen la intención de cada *string*.

[18] concepto de programación en el que un código funciona correctamente al ser ejecutado simultáneamente en múltiples hilos o procesos.

[19] práctica muy extendida en la programación, en la que un dato se introduce directamente en el código fuente en lugar de obtenerlo de un origen como un fichero de recursos.

Gracias a un paquete llamado **Flask-BabelEx** (es posible instalarlo a través del comando `pip install flask-babelex`), es posible sustituir todos los *tokens* por sus correspondientes traducciones en el idioma deseado. Para ello, los identificadores a traducir han de formar parte de los parámetros de la función `gettext`.

Este paquete incluye una utilizada llamada **pybabel**, encargada de buscar entre todos los ficheros del código fuente en busca de identificadores, para así crear un archivo con todos los *tokens* a traducir. Para que esta utilidad funcione correctamente, es necesario indicar el listado de ficheros sobre los que ha de buscar. Para ello, es necesario crear un archivo llamado `babel.cfg` en el directorio principal de la aplicación.

```
[python: htdocs/**.py]
[jinja2: htdocs/templates/**.html]
extensions=jinja2.ext.autoescape,jinja2.ext.with_
```

Figura 8.12: Contenido del fichero **babel.cfg**

Importante: todos los comandos mostrados a continuación hay que ejecutarlos en el directorio raíz de la aplicación (`/var/www/photonics/`), y es necesario ejecutarlos dentro del entorno virtual de *Python* existente. Para acceder al entorno virtual, es necesario ejecutar, en el mismo directorio, el comando `. env/bin/activate`. Es posible salir del entorno virtual ejecutando en cualquier momento el comando `deactivate`.

Para generar este fichero de traducciones, hay que ejecutar el comando `pybabel extract -F babel.cfg -o messages.pot -k BabelText .`. Se creará así un fichero `messages.pot` con todas las cadenas identificadas a través de `gettext` y de `BabelText` (esta última función se explicará posteriormente).

En este punto, se dispone de un fichero genérico de cadenas, que no está asociado a ningún idioma. Es necesario crear un fichero similar para cada uno de los idiomas a los que se desee traducir la aplicación, utilizando el comando `pybabel init -i messages.pot -d babel -l [language]`, siendo `[language]` el código del idioma deseado. Esta instrucción creará, en el directorio `babel`, un fichero `.po` para cada idioma deseado con el listado de `tokens` listo para ser traducidos.

Nota: si fuese necesario ampliar el fichero de traducciones de un idioma después de su creación, es posible hacerlo a través de la instrucción `pybabel update -i messages.pot -d babel -l [language] --ignore-obsolete`.

Una vez traducidos todos los *tokens* de los ficheros *po*, es necesario compilarlos a catálogos binarios de formato `.mo`, utilizando el comando `pybabel compile -d babel`. Este último comando es necesario ejecutarlo cada vez que se quiera modificar el catálogo de traducciones.

Para que la aplicación reconozca el nuevo idioma añadido, es necesario editar el contenido de la clase `AppConfig` del fichero `config.py`, situado en la raíz de la aplicación. En esta clase hay una variable `AVAILABLE_LANGUAGES`, la cual es un *array* con el listado de idiomas existentes en la aplicación. Únicamente hay que

añadir una nueva tupla a la lista, de la forma `('language_code', 'language_name')`. Finalmente, en las variables `RECOVER_PASSWORD_SUBJECT`, `RECOVER_PASSWORD_CONTENT`, `NEW_USER_SUBJECT` y `NEW_USER_CONTENT` del mismo fichero se encuentran las cadenas de texto utilizadas en el envío de correos electrónicos automatizados. Es necesario completar también las cadenas de texto en el nuevo idioma definido.

Con los ficheros de traducción creados, es necesario indicarle al paquete *Flask-BabelEx* el idioma en el que ha de mostrar la vista. El sistema se mostrará por defecto en español, y cualquier usuario podrá modificar a inglés en sus preferencias. En cada petición es necesario comprobar el idioma a utilizar, para lo cual este paquete incluye la función `localeselector`, que se ejecuta al realizarse una petición:

```
@_babel.localeselector
def get_locale():
  if current_user.is_authenticated:
    return current_user.language
  return config.AppConfig.DEFAULT_LANGUAGE
```

Figura 8.13: Selector de idioma a utilizar en cada petición de usuario

Una de las zonas de código que incluye estos *tokens* de traducción son los formularios. Sin embargo, estos se crean previamente a que exista el contexto de una petición. Por lo tanto, no es posible utilizar la función `gettext`, ya que aún no se ha ejecutado la función `localeselector` y, por tanto, no se ha definido ningún idioma. Para solucionarlo, se ha creado la clase `BabelText`. Su funcionamiento es muy sencillo, simplemente aprovecha la forma en la que *Flask-WTF* renderiza mensajes. A la hora de mostrarlos por pantalla, éstos son convertidos a *strings*, lo que produce la ejecución del método `__str__`. Es en este momento ya existe un contexto asociado a una petición y a un usuario, por lo que es posible ejecutar la función `gettext`.

Hay que tener en cuenta que cualquier dato creado desde la aplicación, como los campos dinámicos de las entradas, no será traducido. Independientemente del idioma escogido por el usuario, dichos datos aparecerán siempre con su valor original.

8.12. Migración de datos

Una vez finalizado el desarrollo de la aplicación, se procede a instalarla en la máquina virtual ofrecida por el servicio informático de la Universidad. Para ello, sólo hay que ejecutar en modo *superusuario* el *script* de instalación creado en el primer paso.

Este *script* instalará y configurará automáticamente el sistema y las herramientas necesarias para la ejecución de la aplicación. Además, se encargará de crear el entorno virtual en el que se ejecutará la aplicación y de descomprimirla, dejándola lista para ser utilizada inmediatamente. Sin embargo, en este punto la aplicación no contendrá ningún elemento, categoría o usuario en la base de datos, por lo que posteriormente será

necesario realizar una migración de datos desde la aplicación anterior, transformando los esquemas utilizados en ella al formato requerido por el nuevo sistema.

Al finalizar el *script* se muestran por pantalla todas las contraseñas generadas automáticamente. Es importante guardar estas contraseñas en un lugar seguro, ya que se necesitarán a la hora de completar la migración de datos.

```
Estas son las contraseñas de acceso a los diferentes servicios del sistema.
Guárdelas en un lugar seguro ya que no será posible recuperarlas más tarde.
        SSH y FTP  - usuario photonics: udDn2457Ytaql08X
        PostgreSQL - usuario postgres: CiSN5ud+SHz6fowvhG7ntH4Ld0GAAe6R
        PostgreSQL - usuario photonics: WCFn1LJ8PKdZdrSZrY6Vc0dnpdo=
El posible modificar la contraseña del usuario photonics desde una conexión
SSH con el comando passwd. Sin embargo, las contraseñas de acceso a la base
de datos PostgreSQL no deben modificarse manualmente en ningún caso.
La dirección IP actual del servidor es 10.211.55.18
Es necesario reiniciar el servidor para que la aplicación sea operativa.
```

Figura 8.14: Mensaje mostrado al completar el *script* de migración
Nota: los datos mostrados son de ejemplo, no corresponden al sistema real.

Después de reiniciar el servidor (esto puede hacerse con el comando `shutdown -r 0`), es necesario ejecutar los *scripts* de migración de la base de datos y de migración de ficheros, en ese orden. Para ello hay que subir ambos *scripts* al servidor, así como el *dump* de la base de datos actual, necesario para realizar la conversión al nuevo esquema relacional., utilizando el protocolo **FTP** incorporado durante la instalación del sistema operativo. Estos *scripts* han sido, probablemente, los más costosos de desarrollar de todo el proyecto, tanto en complejidad como en tiempo de desarrollo, por lo que se detallan y explican extensamente en los apéndices **F.2** (**Migración de datos**) y **F.3**, (**Migración de material**).

Es recomendable ejecutar estos *scripts* dentro del mismo entorno virtual de la aplicación, ejecutando el comando `cd /var/www/photonics/ && . env/bin/activate`. Tras la ejecución de cada uno de los *scripts* se crearán sendos ficheros con inserciones **SQL**, tal como se describe en los apéndices correspondientes a ambos. Es posible ejecutar estos ficheros en la propia base de datos utilizando el comando `sudo -u postgres psql -d photonics -f fichero.sql` e introduciendo la contraseña del usuario `photonics` de la base de datos, generada en la instalación.

Una vez ejecutados ambos *scripts* y lanzadas las consultas correspondientes en la base de datos, se puede considerar que el desarrollo y la puesta en producción del sistema están completos. En ese momento, sería interesante que el sistema informático realice un *snapshot*[20] del estado de la máquina virtual, además de modificar las entradas *DNS*[21] del dominio **fotonica.unavarra.es** para que apunten al nuevo servidor.

[20] copia del estado actual de la aplicación, incluyendo su configuración y todos sus ficheros
[21] protocolo de resolución de nombres, el cual asocia un dominio a una dirección IP concreta.

9. Conclusiones y líneas futuras

El proyecto original comenzó como un sistema de gestión sencillo del inventario de un almacén pequeño. La evolución que ha sufrido en esta versión permite que pueda llegar a implementarse en otros almacenes de la Universidad de forma sencilla, gracias al dinamismo que ofrecen varias de las herramientas implementadas, como las herramientas de búsqueda o la posibilidad de modificar la información del inventario de forma dinámica.

Así mismo, está pensado desde un principio con la **escalabilidad** como una de sus principales virtudes. Una posible línea de expansión sería tratar esta aplicación, no como un gestor pensado para un único almacén, sino como un gestor centralizado de múltiples almacenes para toda la Universidad, o incluso para una pequeña empresa.

Sería interesante realizar un análisis para comprobar la carga máxima que puede soportar un pequeño servidor con esta aplicación alojada, tanto en número de conexiones simultáneas como en capacidad de almacenamiento de datos.

La estructura sobre la que se basa esta aplicación es muy sencilla, contando con un único servidor, que realiza de forma simultánea las tareas de servidor web y de base de datos. En caso de una expansión masiva del servicio sería necesario tener en cuenta ciertas medidas para mantener y asegurar tanto el *uptime*[22] como la velocidad de acceso al sistema:

- Replicación del servicio web entre diferentes servidores clones y centralizando las peticiones a uno o varios balanceadores de carga, encargados de repartir de forma eficiente las peticiones de los usuarios entre todos estos servidores.
- Independización del servidor de bases de datos en un sistema externo al de los servidores web, y replicación del mismo en un *cluster* de servidores, asegurando siempre la integridad de los datos almacenados.
- *Cacheo*[23] de la información existente en la base de datos, utilizando para ello un servicio especializado como **Memcached** [37] para reducir la carga de trabajo del servidor de bases de datos en peticiones de sólo lectura.
- Uso de un servicio de búsqueda especializado como **Sphinx** [38] a la hora de realizar búsquedas *full-text* en lugar de confiar por completo en la base de datos.
- Utilización de una base de datos *NoSQL* como **MongoDB** [13] para almacenar en forma de documentos las información y los campos dinámicos de todas la entradas, al estar más orientadas a almacenar información con una estructura menos estricta.

[22] forma de medir el tiempo que una máquina ha estado encendida y operativa de forma ininterrumpida, utilizada normalmente para comprobar la estabilidad y fiabilidad del servicio

[23] almacenamiento de datos de acceso recurrente y de baja frecuencia de actualización para acceder a ellos de forma rápida

Estas medidas no se han implementado ya que la complejidad del proyecto aumentaría de forma exponencial, tanto en tiempo de desarrollo como en *hardware* y recursos necesarios. Además, el uso que va a recibir esta aplicación no justifica en ningún caso la creación de una estructura tan compleja.

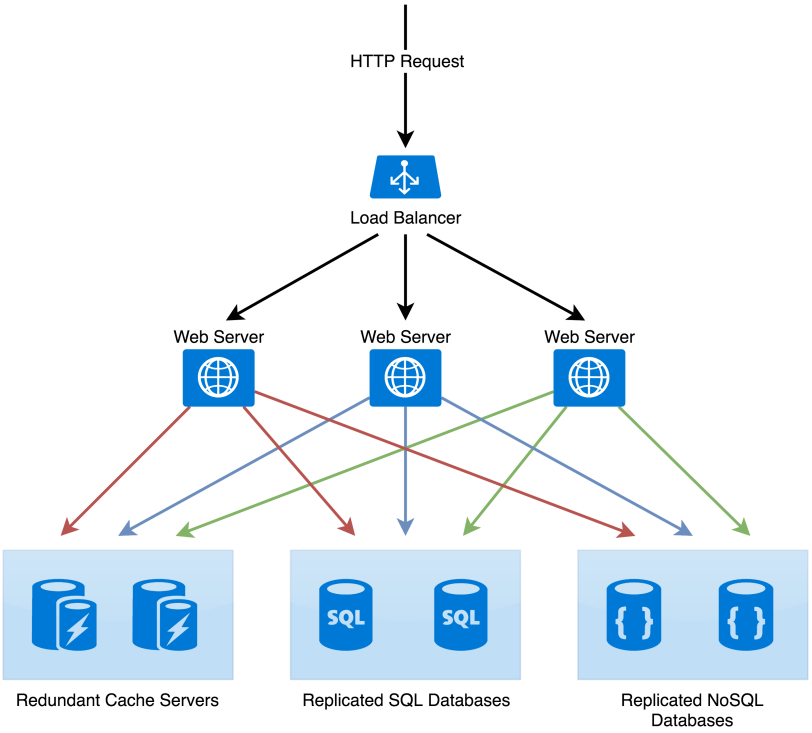


Figura 9.1: Posible estructura escalable y especializada de la aplicación

Una última línea de mejora posible sería el desarrollo de un sistema de división para las entradas. Analizando la base de datos del sistema actual se observa que existen ciertos elementos en ella, que, aunque considerados como un único elemento por la aplicación, existen físicamente como múltiples elementos iguales, por lo que una opción sería poder estructurar una misma entrada en múltiples *subentradas* que, aunque iguales en principio, pueden poseer distintos datos en sus campos.

100404 MINI CIRCUITS MICROWAVE AMPLIFIER

CATEGORÍA	Componentes rf/microondas
ESTADO	OK
FECHA DE ENTRADA	22/09/2009
NÚMERO DE SERIE	zhl-1042j
RESPONSABLE	Alayn Loayssa Lara
DOCUMENTACIÓN	–
DESCRIPCIÓN	Amplificador 10-4200MHz
COMENTARIOS	Hay 2 (había tres, uno se rompió)

Figura 9.2: Entrada con múltiples elementos físicos diferentes en el almacén

10. Bibliografía

[1] *KVM*. URL: http://www.linux-kvm.org/.
[2] *Proxmox*. URL: https://www.proxmox.com/.
[3] *GNU/Linux*. URL: https://www.kernel.org/.
[4] *Debian*. URL: https://www.debian.org/.
[5] *CentOS*. URL: https://www.centos.org/.
[6] *RHEL*. URL: https://www.redhat.com/.
[7] *PHP*. URL: https://www.php.org/.
[8] *Node.js*. URL: https://www.nodejs.org/.
[9] *Python*. URL: https://www.python.org/.
[10] *MySQL*. URL: https://www.mysql.org/.
[11] *MariaDB*. URL: https://www.mariadb.org/.
[12] *PostgreSQL*. URL: https://www.postgresql.org/.
[13] *MongoDB*. URL: https://www.mongodb.com/.
[14] *Apache*. URL: https://httpd.apache.org/.
[15] *Nginx*. URL: https://nginx.org/.
[16] *CGI*. URL: https://www.w3.org/CGI/.
[17] *WSGI*. URL: https://www.python.org/dev/peps/pep-3333/.
[18] *uWSGI*. URL: https://uwsgi-docs.readthedocs.io/.
[19] *Django*. URL: https://www.djangoproject.com/.
[20] *Flask*. URL: http://flask.pocoo.org/.
[21] *Jinja2*. URL: http://jinja.pocoo.org/.
[22] *Flask-SqlAlchemy*. URL: http://flask-sqlalchemy.pocoo.org/.
[23] *SqlAlchemy*. URL: https://tools.ietf.org/html/rfc2616.
[24] *Flask-Login*. URL: https://flask-login.readthedocs.io/.
[25] *Flask-WTF*. URL: https://flask-wtf.readthedocs.io/.
[26] *WTForms*. URL: http://wtforms.readthedocs.io/.
[27] *Flask-BabelEx*. URL: https://pythonhosted.org/Flask-BabelEx/.
[28] *Bootstrap*. URL: http://getbootstrap.com/.
[29] *Gentelella*. URL: https://github.com/puikinsh/gentelella.
[30] *jQuery*. URL: https://jquery.com/.
[31] *Opentip*. URL: http://www.opentip.org/.
[32] *Clipboard.js*. URL: https://clipboardjs.com/.
[33] *SELinux*. URL: http://selinuxproject.org/.
[34] *Let's Encrypt*. URL: https://letsencrypt.org/.
[35] *RFC 2616*. URL: https://tools.ietf.org/html/rfc2616.
[36] *RFC 7230*. URL: https://tools.ietf.org/html/rfc7230.
[37] *Memcached*. URL: https://memcached.org/.
[38] *Sphinx*. URL: http://sphinxsearch.com/.

A. Manual de uso para usuarios

A.1. Autenticación

Al entrar a la aplicación por primera vez se accede al área de autenticación. Es necesario introducir un nombre de usuario y contraseña válidos para acceder al sistema.

Figura A.1: Formulario de autenticación de usuarios

En caso de haber olvidado la contraseña, es posible restaurarla sin involucrar a un administrador. Para ello, se ofrece una zona donde, al introducir el correo electrónico del usuario, se enviará un *email* a dicha dirección con una nueva contraseña generada de forma aleatoria, la cual podrá ser modificada posteriormente por el usuario. Por seguridad, el formulario de recuperación de contraseña no indica si la dirección de correo electrónico introducida corresponde a un usuario real de la aplicación.

Figura A.2: Formulario de restauración de contraseña

A.2. Página de inicio

Una vez autenticado en el sistema, se mostrará la página inicial de la aplicación. Ésta se encuentra dividida en tres zonas: la barra de navegación lateral, el menú superior y el cuerpo principal de la aplicación.

En la barra de navegación lateral aparecerá el árbol de categorías del inventario, detallado más adelante. En la zona superior se pueden encontrar los botones de acceso directo a los sistemas de búsqueda de inventario, así como las herramientas de personalización de perfil del usuario.

En esta página principal se observará que en el cuerpo de la aplicación se muestra también el listado de categorías de inventario.

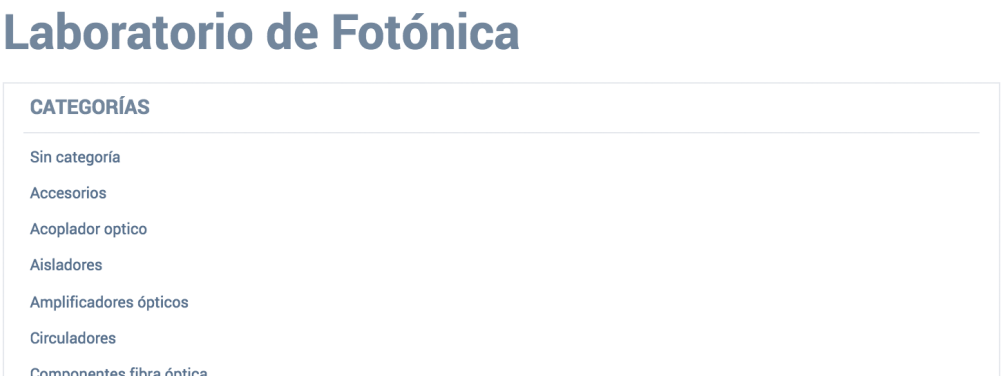


Figura A.3: Contenido de la página de inicio de la aplicación

A.3. Perfil

En la zona derecha del menú superior de la aplicación se puede observar el nombre del usuario activo actualmente. Pulsando sobre él, es posible cerrar su sesión o acceder a las opciones de personalización de su perfil.

Figura A.4: Acceso al área de perfil de usuario desde el menú superior

Dentro del área de perfil se pueden ver todos los datos del usuario activo, pero no es posible modificarlos. Únicamente los usuarios con credenciales de administración

pueden modificar los datos básicos del usuario, como su nombre o su correo electrónico.

En esta zona se muestran también dos formularios. El primero de ellos permite modificar el idioma en el que se mostrará la aplicación para el usuario, pudiendo elegir entre español e inglés. En el segundo formulario es posible modificar la contraseña actual del usuario. A la hora de modificar la contraseña, es necesario escribir también la contraseña actual como medida de seguridad.

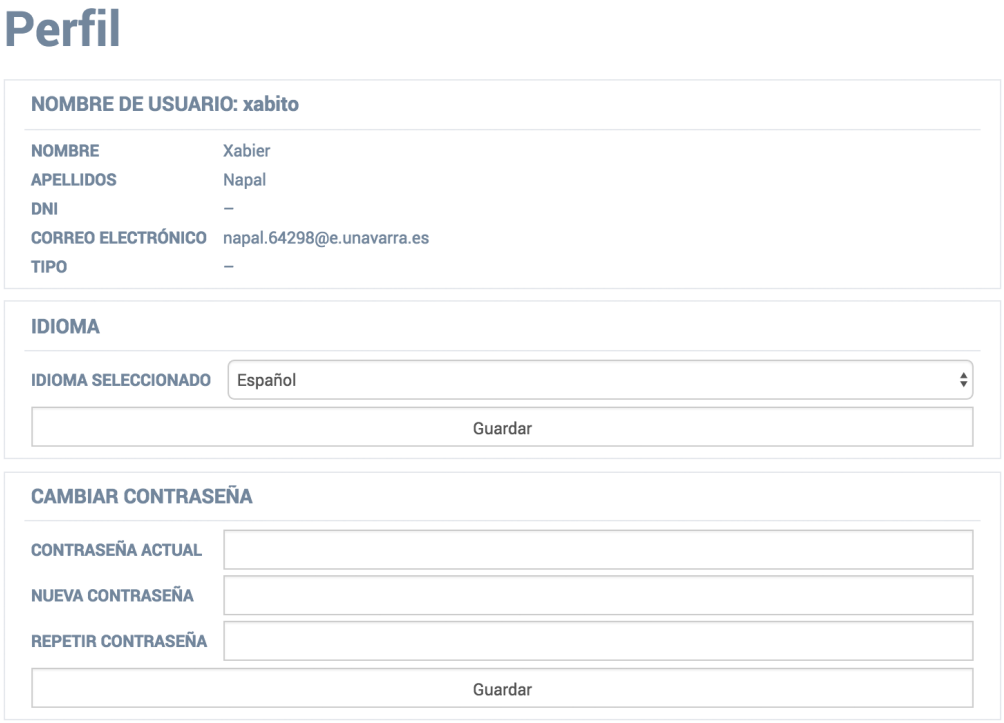


Figura A.5: Área de perfil de usuario

A.4. Inventario

El área de inventario contiene la mayor parte de la funcionalidad de la aplicación. Prácticamente la totalidad de la interfaz hace referencia a alguna herramienta de este área. En la barra lateral de navegación se encuentra el listado de categorías y en la parte izquierda del menú superior están los accesos a los sistemas de búsqueda.

A.4.1. Categorías

Pueden visualizarse los elementos de una categoría pulsando sobre ella, tanto el barra lateral como en el desplegable mostrado en la página principal de la aplicación.

Al acceder a una categoría se visualizará el listado completo de elementos asociados a dicha categoría. Estos elementos se muestran en bloques desplegables contraídos por defecto, expandibles pulsando sobre ellos.

Figura A.6: Área de inventario con el listado de elementos de una categoría

En esta visual no se muestran todos los datos relativos a una entrada del elemento. Únicamente aparecen su identificador, su nombre, su categoría, su estado, su descripción, sus comentarios y, en caso de existir, los datos de su uso. También aparecen el listado de archivos digitales asociados, en caso de contener alguno.

En la zona superior de cada una de las entradas aparecen tres botones. El primero de ellos, con forma de ojo, sirve para acceder a la información detallada de ese elemento. El segundo de ellos permite enviar al portapapeles del sistema la *URL* correspondiente a esa entrada. El último de ellos tiene como finalidad ocultar y visualizar la información de la entrada.

Figura A.7: Botones situados en cada elemento del listado de entradas

A.4.2. Detalle de una entrada

Al acceder a la información detallada de una entrada a través del botón existente para ello, se muestra una visual similar, pero con el resto de datos asociados a la misma no visibles en la anterior pantalla.

Figura A.8: Detalle de la información de un elemento del inventario

En esta zona los botones mostrados en la entrada son distintos. El primero de ellos permite volver a la visual de categorías. Además, aparecen dos nuevos botones más, que permiten editar la información de una entrada, así como eliminarla del sistema.

Precaución: no es posible deshacer el borrado de una entrada. En el momento en el que se elimine, no será posible recuperarla del sistema.

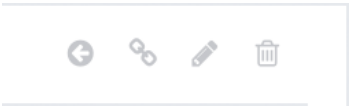


Figura A.9: Botones visibles en la interfaz de detalle de un elemento

A.4.3. Inserción y edición de entradas

Es posible editar una entrada desde el botón disponible en su visual detallada. También es posible crear elementos nuevos desde el botón ofrecido en el menú superior de la aplicación. Para poder realizar cualquiera de las dos tareas, es necesario disponer de un usuario con permisos de escritura en la aplicación.

Ambas funcionalidades muestran el mismo formulario en pantalla, con una visual similar a la del detallado de una entrada, pero permitiendo modificar cualquiera de sus campos. Así mismo, incluyen la posibilidad de subir ficheros e imágenes y asociarlos a la entrada. Esta subida de ficheros puede realizarse, si el navegador provee de la funcionalidad *drag&drop*, a través del arrastre de los mismos.

Importante: como novedad respecto a la aplicación antigua, ahora es posible realizar la subida de varios ficheros diferentes a una misma entrada, por lo que ya no es necesario comprimirlos en archivos *zip* antes de subirlos a la aplicación.

Figura A.10: Interfaz de inserción y modificación de entradas

Importante: la metodología de creación de nuevas entradas ha variado respecto a la plataforma antigua. En dicha versión, primero se reservaba un número o identificador único para la entrada, y posteriormente se rellenaba toda su información. En la nueva aplicación, este identificador no es generado hasta haber completado y rellenado todos los datos de la entrada por primera vez.

A.4.4. Búsqueda simple

En la zona izquierda del menú superior de la aplicación se encuentra un campo de texto para acceder al sistema de búsqueda simple. Con insertar en valor de búsqueda deseado y pulsar sobre el botón de búsqueda, se redirigirá al usuario a una página con una visual similar a la del listado de categorías, pero con los resultados que coincidan con la búsqueda.

Figura A.11: Formulario de búsqueda simple

A.4.5. Búsqueda avanzada

También en la parte izquierda del menú superior de la aplicación, se encuentra otro botón para acceder al sistema de búsqueda avanzada.

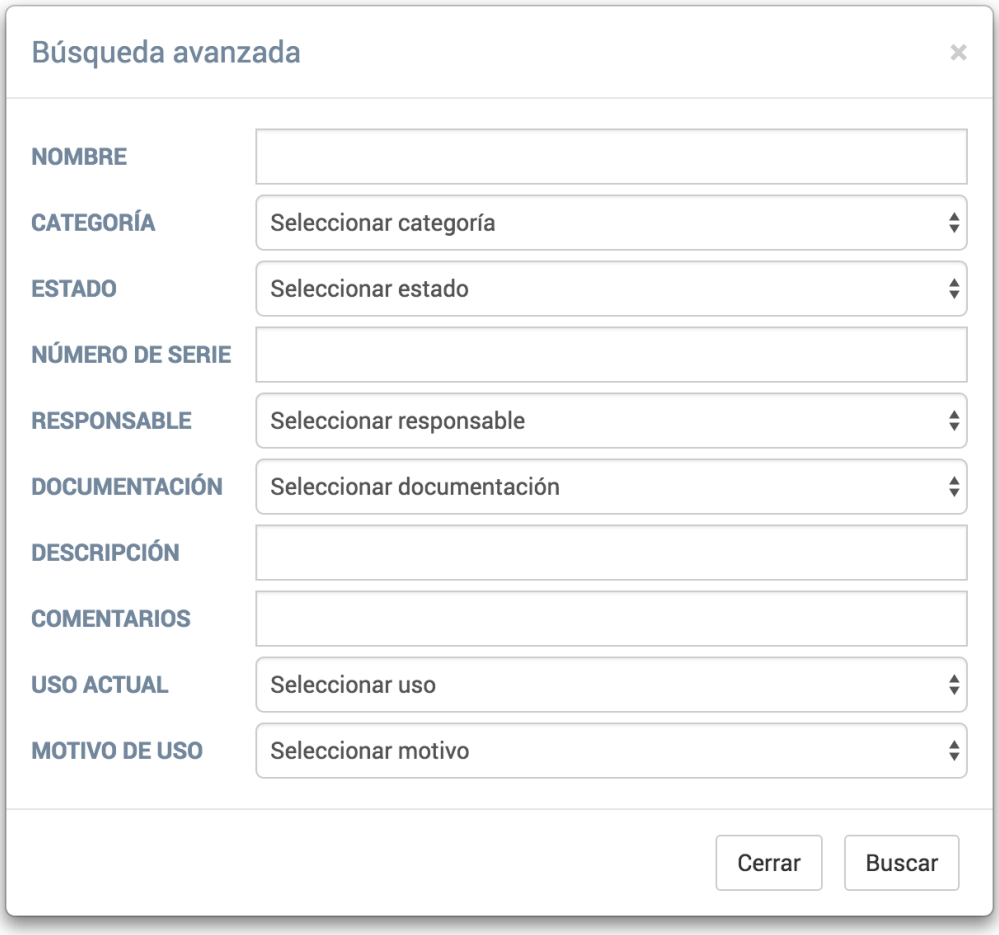


Figura A.12: Panel de búsqueda avanzada

A diferencia del sistema simple de búsqueda, a través de este módulo es posible filtrar los resultados a buscar por cualquiera de los campos existentes en una entrada, incluyendo también los campos dinámicos asociados a la misma, tanto los generales como los asociados a categorías.

B. Manual de uso para administradores

Aquellos usuarios con permisos de administración podrán comprobar que en el menú superior de la aplicación tienen un botón extra para acceder al panel de administración.

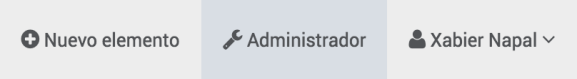


Figura B.1: Menú superior con acceso al panel de administración

El área de administración está dividida en diferentes apartados que afectan a distintas áreas de la aplicación. En cada uno de los posteriores apartados se explicarán las funcionalidades relacionadas a cada uno de estos apartados.

B.1. Inventario

Este área se divide a su vez en cuatro zonas, las cuales permiten modificar los datos disponibles para varios campos de cada entrada del inventario: los selectores de estado, documentación y uso, así como el listado de campos dinámicos asociados a todos los elementos. Estas cuatro zonas se utilizan de la misma manera, y permiten añadir nuevos elementos, modificarlos o eliminarlos directamente, pulsando en los diferentes botones disponibles.

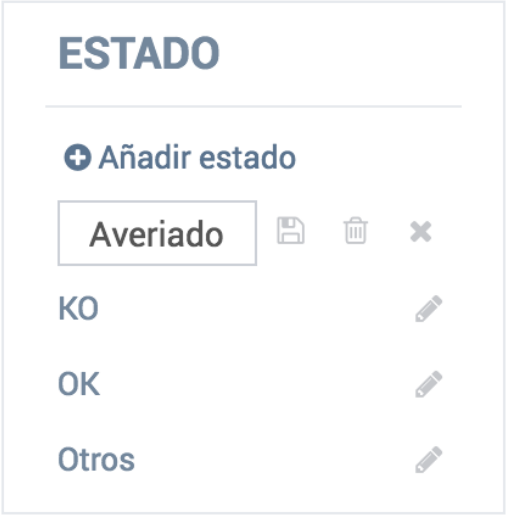


Figura B.2: Área de inventario del panel de administración

B.2. Usuarios

Este área se divide en dos zonas distintas, aquella que permite añadir o modificar campos al selector de categorización de usuarios, y que funciona exactamente igual a las descritas en el apartado anterior, y la encargada de la gestión de los usuarios de la aplicación. En esta última es posible, a través de un completo formulario, modificar todos los datos relativos a un usuario, gestionar sus permisos de acceso y permitiendo incluso modificar su contraseña en caso de ser necesario.

Figura B.3: Formulario de gestión de usuarios del panel de administración

Al crear un nuevo usuario, éste recibirá un correo electrónico a la dirección especificada en el formulario con sus credenciales de acceso: su usuario y su contraseña. Este usuario sólo podrá acceder a la aplicación si dispone del permiso **Activo**. De esta forma es posible desactivar usuarios temporalmente sin necesidad de borrarlos.

Nota: por seguridad, no es posible eliminar el usuario activo.

Importante: al eliminar un usuario sus datos relacionados desaparecerán: todas las entradas cuyo responsable sea este usuario quedarán sin responsable, y si el usuario se encuentra utilizando alguna de las entradas, ésta quedará sin uso.

B.3. Categorías

El área de categorías comprende dos apartados diferentes. El primero de ellos es la modificación del árbol de categorías, permitiendo añadir o modificar categorías en cualquier nivel del árbol. No hay un límite en el número de subcategorías que es posible anidar, aunque por claridad no se recomienda que el árbol contenga más de tres niveles. Esta zona funciona de igual forma a las descritas anteriormente.

Importante: en caso de eliminar una categoría existente que posea entradas asociadas, estos elementos no desaparecerán del inventario. En su lugar, se reubicarán en una pseudo categoría. Ya que normalmente un elemento ha de tener obligatoriamente una categoría asociada, sólo se podrán realizar modificaciones sobre estos elementos si a su vez se les asigna una nueva categoría.

La segunda zona permite modificar los campos dinámicos asociados a una categoría concreta. Se accede a ella a través del botón de edición una categoría. Desde este *popup* es posible añadir, modificar o eliminar campos asociados a una categoría concreta, así como modificar el nombre de la propia categoría o eliminarla.

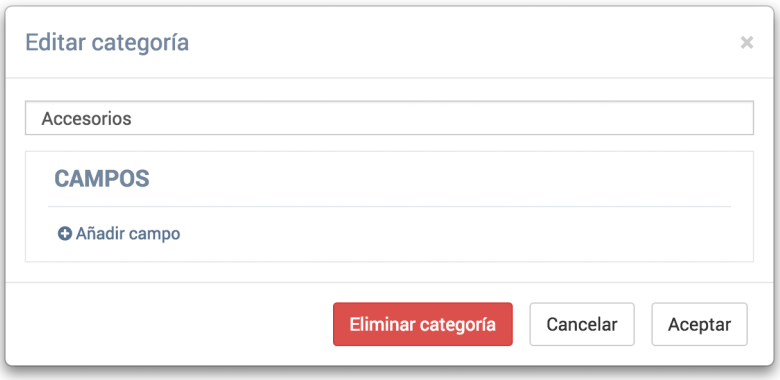


Figura B.4: *Popup* de gestión de campos asociados a categorías

Importante: es necesario tener en cuenta que, al eliminar un campo asociado a una categoría, todas las entradas pertenecientes a la misma perderán ese campo, incluso aquellas que tuviesen información almacenada en el mismo.

B.4. Notificaciones

Una de las funcionalidades de la aplicación es la posibilidad de mostrar notificaciones los usuarios de la misma. Estas notificaciones se dividen en tres tipos, dependiendo de la zona en la que son visibles: la página de autenticación, las vistas de visualización de datos del inventario y las páginas de inserción y modificación de elementos.

B.5. Copias de seguridad

A través de este menú es posible realizar copias de seguridad de forma manual, tanto de la base de datos completa como de los ficheros y material asociado a las entradas del inventario. También permite recuperar las copias de seguridad realizadas previamente.

Los *backups* generados no siguen ningún formato estándar, sino que se han diseñado para ser utilizados a través de este panel, y es el único formato válido aceptado por los mismos. En caso de restaurar un *backup*, es importante asegurarse que el fichero subido corresponda con un fichero de *backup* creado con la aplicación, para evitar posibles problemas.

Figura B.5: Área de *backup* del panel de administración

La creación de un *backup* automático se realizará a través de una petición a la ruta `http://fotonica.unavarra.es/admin/backup/auto/<username>:<password>`, sustituyendo los campos `<username>` y `<password>` por las credenciales de un usuario de la aplicación con permisos de administrador. Esta petición retornará un fichero **zip** con el *backup* de la base de datos y de los ficheros. Los ficheros generados en este *backup* son totalmente compatibles con los generados manualmente, por lo que es posible restaurar estos *backups* a través de la interfaz del panel de administración.

B.6. Actualizaciones

Permite gestionar la instalación de actualizaciones en el sistema operativo directamente desde la interfaz web. Consta de un único botón, encargado de realizar la búsqueda de actualizaciones pendientes. Una vez la búsqueda finalice, se mostrará un *popup* al administrador, indicando el número de actualizaciones existentes en el sistema, ofreciendo la posibilidad de realizar una actualización completa.

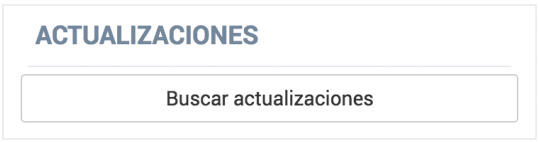


Figura B.6: Área de gestión de actualizaciones del panel de administración

Cualquier actualización del sistema a través de este método es responsabilidad única del administrador. Es importante tener en cuenta que cualquier actualización puede provocar inestabilidad o fallo del sistema, por lo que es recomendable hacer una copia de seguridad completa de la máquina antes de realizar el proceso de actualización.

C. Manual de instalación de la aplicación

En este apéndice se va a definir el proceso de la primera instalación de la aplicación, incluyendo la migración de datos de la plataforma anterior. Se partirá de un sistema operativo recién instalado y completamente limpio. Pueden obtenerse más instrucciones sobre cómo instalar el sistema operativo en el apéndice **E** (**Instalación del sistema operativo CentOS 7**).

Para realizar la instalación de la aplicación es necesario acceder al sistema como usuario administrador, o `root`. El primer paso será la ejecución del *script* de instalación. Este *script* será el encargado de instalar todas las herramientas y utilidades necesarias para el correcto funcionamiento del sistema, así como de descomprimir la aplicación y crear la base de datos, dejándola lista para ser utilizada. Es necesario asegurarse de que el *script* cuenta con permisos de ejecución para poder ser llamado desde la consola de comandos, utilizando la instrucción `chmod +x install.sh`.

Para ejecutar el *script*, basta con introducir `./install.sh` en el directorio donde se encuentre. En la captura de pantalla que se muestra a continuación, el comando introducido posee otros parámetros. Éstos permiten guardar en un fichero de texto la información que muestre el *script* por pantalla, para así facilitar el *debug* del proceso de instalación, en caso de ocurrir algún fallo. Se recomienda, para facilitar el guardado de información importante que se muestre por pantalla, escribir el comando completo:

```
./install 2>&1 | tee install_output.log
```

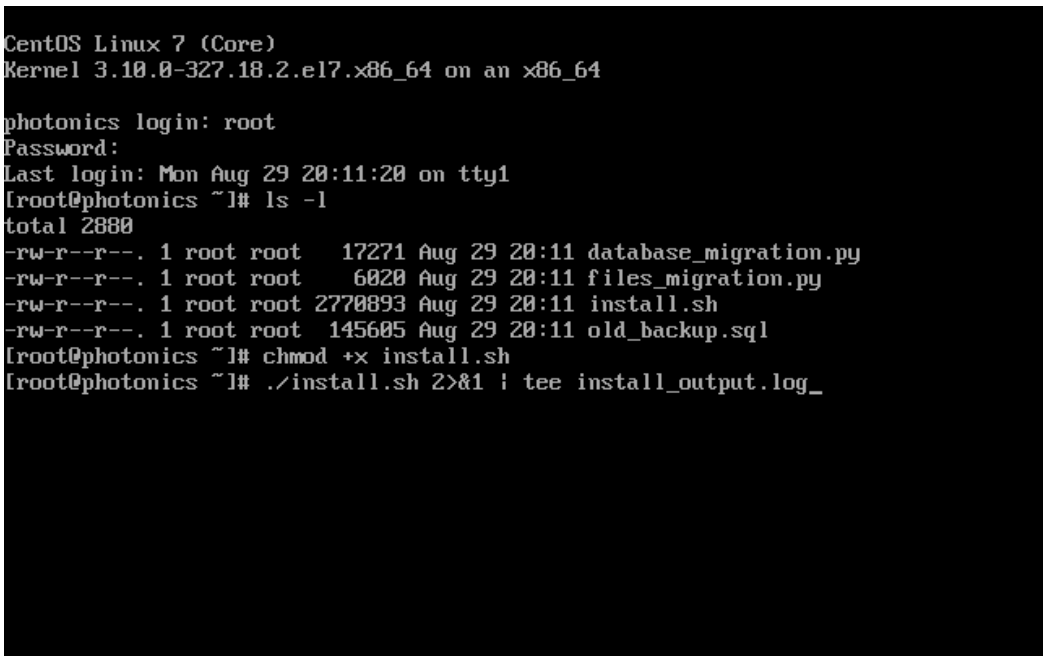


Figura C.1: Ejecución del *script* de instalación

Una vez finalizada la instalación, aparecerán por pantalla un listado de contraseñas. Es importante almacenarlas en lugar seguro ya que, serán necesarias en los siguientes pasos de la instalación, así como en caso de cualquier problema en el futuro. Será necesario reinicar la aplicación para continuar con los siguientes pasos, utilizando el comando `reboot`.

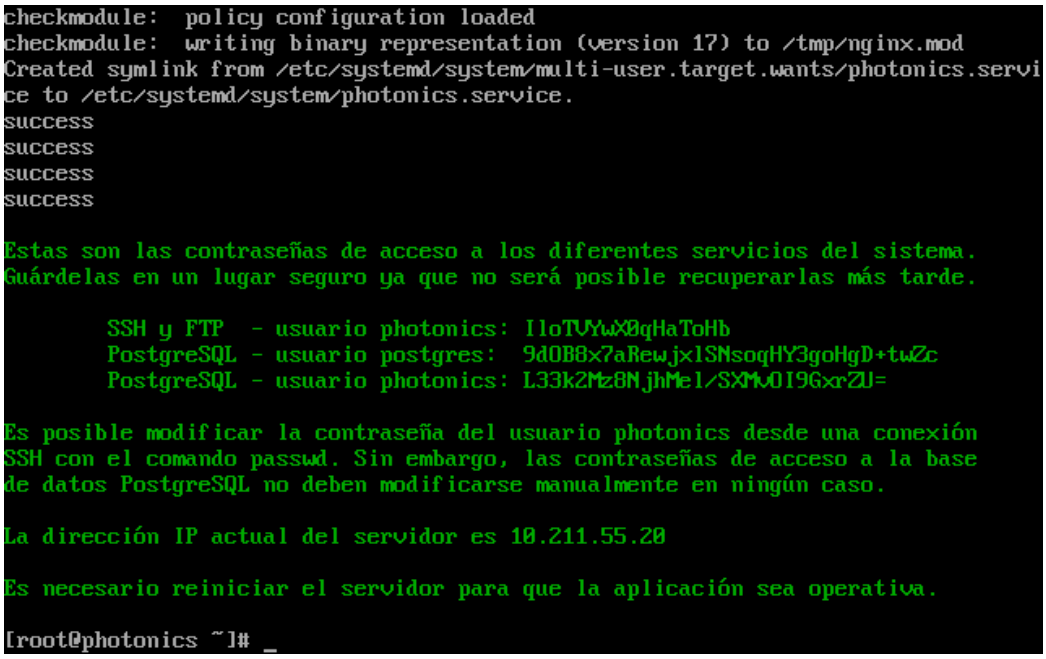


Figura C.2: Contraseñas generadas por el *script* de instalación
Nota: las contraseñas no se corresponden a las utilizadas en el sistema final.

Desde este momento, la aplicación será accesible a través de un navegador, especificando la *IP* del servidor. Esta *IP* puede conocerse observando la información mostrada en el *script* de migración, o ejecutando el comando `ip addr`. Sin embargo, la aplicación no contendrá datos de usuarios, por lo que no será posible autenticarse en ella.

A continuación se procederá a migrar la base de datos de la aplicación antigua. Para ello es necesario contar con un *dump SQL* de la misma, el cual se puede conseguir en el último *backup* creado por el administrador del sistema. Se tratará este archivo con el *script* `database_migration.py` creado para este fin. Su sintaxis es la siguiente:
`python database_migration.py old_backup.sql db.sql`

Una vez generado el *script* de importación de datos, será necesario insertarlo en la base de datos, utilizando el comando `sudo -u postgres psql -d photonics -f db.sql`. Al introducir esta instrucción, se solicitará una contraseña. Ésta se corresponde a la indicada en la línea `PostgreSQL - usuario postgres` del resultado del *script* de instalación.

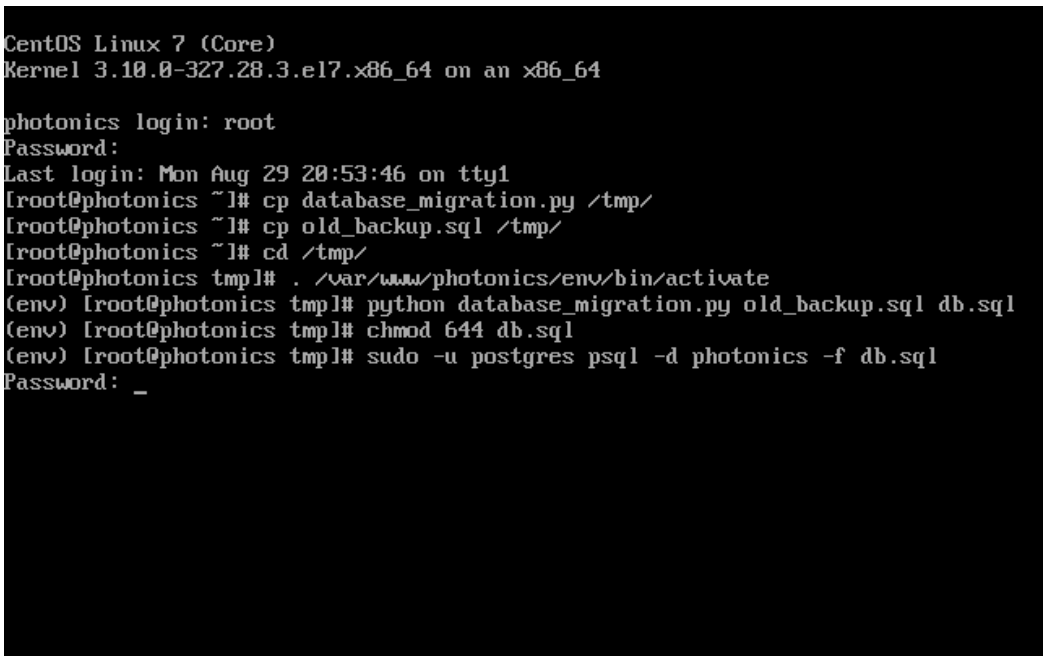


Figura C.3: Ejecución del *script* de migración de datos

Finalmente, será necesario importar los ficheros digitales existentes en la anterior plataforma, utilizando el *script* `files_migration.sql`. Para la ejecución de este *script* es indispensable poder acceder a la red interna de la Universidad y a la *IP* `172.18.67.26`, correspondiente al sistema antiguo. Con el *script* finalizado, sólo queda importar a la base de datos el fichero *SQL* generado. Esto se hará de la misma forma que se ha hecho previamente con el *script* de migración de la base de datos.

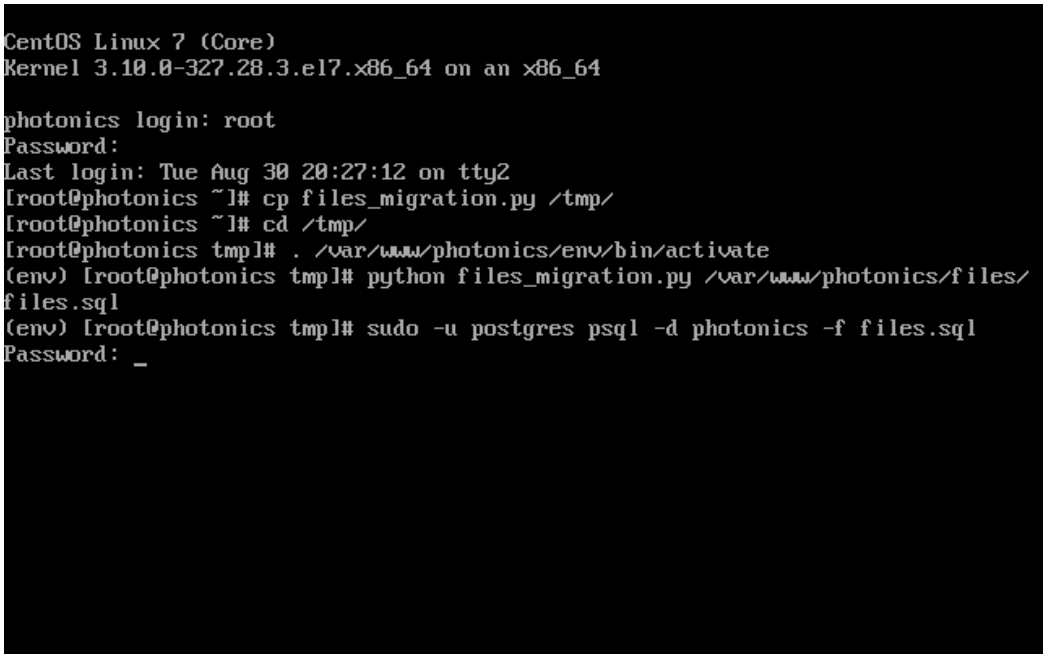


Figura C.4: Ejecución del *script* de migración de archivos

D. Benchmark de funciones de derivación

Nota: este script sólo puede ejecutarse a partir de la versión 3.3 de Python, y es necesario disponer del paquete **werkzeug** en el sistema (puede instalarse con el comando `pip install werkzeug`).

D.1. Script en Python

```
# -*- coding: utf-8 -*-
import time
import functools
from werkzeug.security import generate_password_hash
def calculate_execution_time(iterations):
  def decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
      execution_time = 0.0
      for i in range(iterations):
        start = time.process_time()
        func(*args, **kwargs)
        end = time.process_time()
        execution_time += (end - start)
      return execution_time / iterations
    return wrapper
  return decorator
def create_password_generator(hash, iters, salt_length):
  def password_generator(password):
    return generate_password_hash(
      password,
      method='pbkdf2:' + hash + ':' + str(iters),
      salt_length=salt_length)
  return password_generator
if __name__ == '__main__':
  password_types = [
    {'hash': 'sha256', 'iters': 100000, 'salt': 16},
    {'hash': 'sha256', 'iters': 100000, 'salt': 32},
    {'hash': 'sha256', 'iters': 100000, 'salt': 128},
    {'hash': 'sha512', 'iters': 100000, 'salt': 16},
    {'hash': 'sha512', 'iters': 100000, 'salt': 32},
    {'hash': 'sha512', 'iters': 100000, 'salt': 128},
    {'hash': 'sha256', 'iters': 250000, 'salt': 16},
    {'hash': 'sha256', 'iters': 250000, 'salt': 32},
    {'hash': 'sha256', 'iters': 250000, 'salt': 128},
    {'hash': 'sha512', 'iters': 250000, 'salt': 16},
```

```
    {'hash': 'sha512', 'iters': 250000, 'salt': 32},
    {'hash': 'sha512', 'iters': 250000, 'salt': 128},
    {'hash': 'sha256', 'iters': 1000000, 'salt': 16},
    {'hash': 'sha256', 'iters': 1000000, 'salt': 32},
    {'hash': 'sha256', 'iters': 1000000, 'salt': 128},
    {'hash': 'sha512', 'iters': 1000000, 'salt': 16},
    {'hash': 'sha512', 'iters': 1000000, 'salt': 32},
    {'hash': 'sha512', 'iters': 1000000, 'salt': 128}
  ]
  for x in password_types:
    generator = make_generator(x['hash'], x['iters'], x['salt'])
    exec_time = calculate_execution_time(100)(generator)('p4ssw0rd')
    print(
      'hash:', x['hash'],
      'iterations:', str(x['iters']).rjust(7),
      'salt:', str(x['salt']).rjust(3),
      'time:', str(exec_time) + 's')
```

D.2. Resultados

```
hash: sha256 iters:  100000 salt:  16 time: 0.4030951466s
hash: sha256 iters:  100000 salt:  32 time: 0.39550737080000004s
hash: sha256 iters:  100000 salt: 128 time: 0.38949276699999996s
hash: sha512 iters:  100000 salt:  16 time: 0.4876650844s
hash: sha512 iters:  100000 salt:  32 time: 0.4894117093999995s
hash: sha512 iters:  100000 salt: 128 time: 0.4902494888000003s
hash: sha256 iters:  250000 salt:  16 time: 0.9695547034000007s
hash: sha256 iters:  250000 salt:  32 time: 0.9740975120000002s
hash: sha256 iters:  250000 salt: 128 time: 0.9762040812000002s
hash: sha512 iters:  250000 salt:  16 time: 1.2168724491999996s
hash: sha512 iters:  250000 salt:  32 time: 1.2195558435999985s
hash: sha512 iters:  250000 salt: 128 time: 1.2203969666000005s
hash: sha256 iters: 1000000 salt:  16 time: 3.911143083199998s
hash: sha256 iters: 1000000 salt:  32 time: 3.8753675282000017s
hash: sha256 iters: 1000000 salt: 128 time: 3.9143436952000004s
hash: sha512 iters: 1000000 salt:  16 time: 4.926544669800004s
hash: sha512 iters: 1000000 salt:  32 time: 5.139746514799993s
hash: sha512 iters: 1000000 salt: 128 time: 5.167807791199993s
```

D.3. Conclusiones

Por lo que se puede comprobar observando los resultados, hay dos factores fundamentales en el tiempo necesario para computar cada función de derivación. El primero de ellos es la función de hasheo utilizada. Esto es acorde a los resultados esperados, ya que

SHA512 es una función diseñada para ser más cara computacionalmente, ya que, entre otras cosas, genera un resultado de 64 bytes y emplea 80 iteraciones, frente a los 32 bytes de salida y 64 iteraciones de *SHA256*.

El segundo factor que incrementa el tiempo de cálculo es el número de iteraciones sobre las que se aplica la función de *hasheo* en las sucesivas claves derivadas. Esto es algo evidente, ya que ese es precisamente el objetivo de este tipo de algoritmos.

Finalmente, se comprueba que incrementar la longitud del *salt* aleatorio no produce un cambio significativo en los tiempos de cómputo. Es posible que con cadenas de mayor longitud el tiempo final se viese incrementado, debido al coste de generar una cadena aleatoria tan larga, así como el hasheo de dichas cadenas. Sin embargo, no parece ser un factor determinante, ya que además de no mejorar la seguridad del sistema, incrementa el coste de almacenamiento del mismo.

En base a los resultados obtenidos, se ha optado por utilizar la función de *hasheo* *SHA512*, derivada con 100000 iteraciones y con una longitud de 16 caracteres en el *salt*. Se ha elegido esta función ya que en teoría es más segura, reduciendo las posiblidades de encontrar un algoritmo capaz de generar colisiones de *hashes*.

De esta forma se ha conseguido un equilibrio en el tiempo de cómputo, siendo lo bastante rápido como para no interferir en el funcionamiento de la aplicación y al mismo tiempo lo suficientemente lento como para complicar un ataque de fuerza bruta. Con un coste aproximado de 0.5s por cadena, es posible generar como máximo 172800 cadenas diariamente. Esto significa que el coste de *crackear* una cadena de siete caracteres alfanuméricos puede ascender a más de 15000 años. Por supuesto, estas cifras se basan en el hardware utilizado en este *benchmark*. Sería posible recortar de forma notable el tiempo de cómputo requerido utilizando *software* específico ejecutado en una granja de *GPUs*, un conjunto de sistemas creado específicamente para el *crackeo* de contraseñas.

E. Instalación del sistema operativo CentOS 7

El proceso de instalación se ha realizado utilizando un software de virtualización, aunque el proceso es el mismo en caso de utilizar un *hardware* real.

Una vez arrancada la máquina con el medio de instalación de **CentOS** inseratdo (normalmente un *DVD* o un dispositivo *USB*), se muestran por pantalla las diferentes opciones de instalación disponibles:

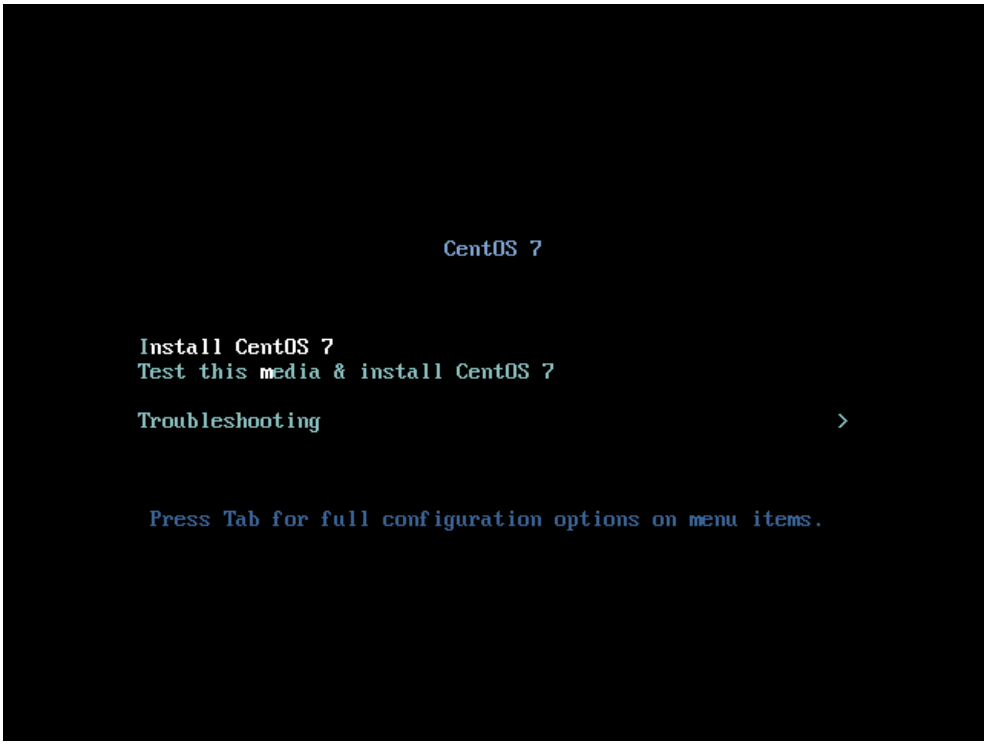


Figura E.1: Menú principal de la instalación de **CentOS 7**

Por defecto, la primera opción, `Install CentOS 7`, ejecuta el gestor de la instalación en modo gráfico. En el caso de la máquina virtual utilizada en la instalación, existen incompatibilidades con la interfaz gráfica de la instalación, por lo que se optará por una instalación tradicional en modo texto. Este paso también puede llegar a ser necesario en caso de sistemas que no cuenten con aceleración gráfica.

Para ejecutar la instalación en modo texto, es necesario pulsar en el teclado la tecla `Tab` o tabulación. Al hacerlo, se mostrarán las opciones de instalación de la opción seleccionada. Solamente es necesario incluir al final la opción `text` y pulsar la tecla `Enter` para comenzar la instalación sin un entorno gráfico.

Figura E.2: Ejecución de la instalación de **CentOS 7** en modo texto

Una vez iniciado el ayudante de instalación, se mostrarán por pantalla las diferentes opciones de configuración del sistema, las cuales se enumeran a continuación:

- `Language settings`: permite seleccionar el idioma en el que se instalará el sistema operativo. Esta opción no influye en el funcionamiento final de la aplicación, por lo que se puede configurar a gusto del administrador del sistema.
- `Timezone settings`: indica la zona horaria principal del sistema. Esta opción tampoco afecta al funcionamiento del sistema, aunque es recomendable configurar el huso horario en el que se vaya a utilizar la aplicación, ya que las fechas almacenadas en la base de datos utilizarán como base la fecha del sistema. En este caso se configurará la zona horaria `Europe/Madrid`.
- `Installation source`: permite seleccionar las fuentes desde las que se obtendrán los paquetes necesarios para instalar el sistema. La mayoría de paquetes utilizados vienen incluidos por defecto en el dispositivo de instalación utilizado, por lo que se optará por una instalacción *offline*, seleccionando la opción `Local media`.
- `Software selection`: es posible instalar esta distribución con distintos conjuntos de paquetes base. Ya que se ha desarrollado un *script* de instalación con todas las herramientas necesarias para este proyecto, se optará por una instalación mínima, a través de la opción `Minimal install`.
- `Installation Destination`: permite modificar la tabla de particiones del disco en el que se instalará el sistema operativo. Para esta instalación se mantendrá

un particionado automático con la opción `Automatic partitioning selected`, aunque es posible que se requiera de otro tipo de particionado en la máquina utilizada en producción.

- `Kdump`: es un mecanismo de volcado de información en caso de errores en el *kernel*. En el caso de la máquina de desarrollo, este se desactivará con la opción `Kdump is disabled`, aunque, nuevamente, el servicio informático es libre de configurarlo en base a sus necesidades en la máquina de producción.

- `Network configuration`: a lo largo de la instalación será necesario poseer acceso, tanto a *Internet* como a la red interna de la Universidad, por lo que es obligatorio configurar correctamente la red, indicando la interfaz y los parámetros de conexión correctos.

- `Root password`: es necesario crear una contraseña para el usuario administrador del sistema. Esta es la única contraseña generada manualmente a lo largo de la instalación, ya que el resto se generarán automáticamente al ejecutar el *script* de instalación de la aplicación.

- `User creation`: permite la creación automática de un usuario adicional durante el proceso de instalación. Ya que los usuarios necesarios se crearán posteriormente a través del *script* de instalación, no es necesario crear ningún usuario en este momento, por lo que se seleccionará la opción `No user will be created`.

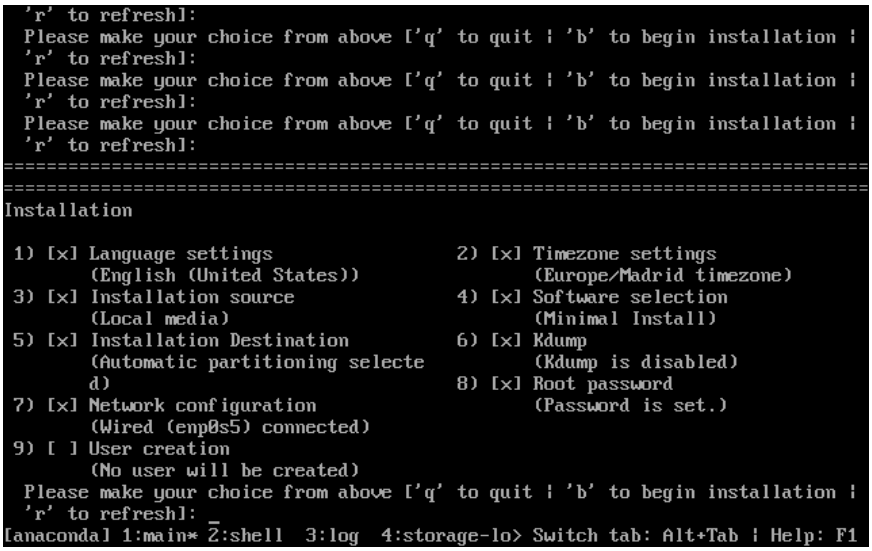


Figura E.3: Configuración de opciones en el ayudante de instalación

Una vez configuradas todas las opciones de instalación necesarias, sólo queda comenzar el proceso pulsando la tecla `b`, con lo que dará comienzo a la instalación. Una vez finalizada, es necesario reiniciar el sistema, y éste estará listo para utilizarse.

F. Scripts utilizados en el desarrollo

F.1. Script de instalación

Nota: ya que la longitud de este *script* es de aproximadamente 400 líneas, no se mostrará su código en este apéndice. Es posible encontrar el *script* completo en el archivo **install.sh** incluido con todos los ficheros del proyecto.

Se ha desarrollado este *script* para facilitar la instalación y configuración del sistema operativo para poder ejecutar la aplicación. Se ha *testeado* en una instalación mínima de *CentOS 7*, por lo que su funcionamiento no está asegurado en otras distribuciones. Es necesario tener permisos de superadministrador en el sistema y disponer de la herramienta **yum** para utilizar este *script*.

A pesar de su extensión, es un *script* relativamente sencillo, encargado principalmente de instalar los paquetes necesarios para ejecutar la aplicación. El funcionamiento del mismo puede dividirse en las siguientes secciones:

- Actualización del sistema a la última versión e instalación de herramientas necesarias para la instalación posterior de paquetes específicos de la aplicación.
- Generación automática de contraseñas para todos los usuarios a crear, tanto aquellos de sistema como los de la aplicación y la base de datos.
- Instalación del intérprete de *Python* en su versión 3.5, así como de *pip*, el gestor de paquetes de *Python* y la herramienta para crear entornos virtuales.
- Instalación de la base de datos *PostgreSQL* en su versión 9.5, creación de los usuarios de la misma y gestión de sus permisos de acceso y contraseñas.
- Instalación del servidor web *Nginx* a través del repositorio oficial, y configuración de las rutas de acceso así como los parámetros necesarios para acceder al *socket* creado a través del protocolo *WSGI*.
- Creación de los directorios y el entorno virtual de la instalación, e instalación de todos los módulos *Python* requeridos para el funcionamiento de la misma, así como descompresión de todos los ficheros de la aplicación en dichos directorios.
- Creación de todas las entidades de la base de datos utilizadas en la aplicación.
- Instalación y configuración de un servidor *FTP* para permitir subidas de archivos de forma directa al sistema.
- Modificación de los protocolos y restricciones de seguridad del módulo de sistema *SELinux* para permitir la correcta conexión y uso de la aplicación.
- Configuración del firewall incluido en el sistema para permitir conexiones externas a los servicios disponibles en el sistema: el servidor web y el servidor FTP.

F.2. Migración de datos

Nota: por su longitud y complejidad, el código del *script* no se incluye en este apéndice. Podrá encontrarse en el fichero `database_migration.py` incluido con el resto de archivos entregados en el proyecto.

Este es, probablemente, el *script* más complejo de todos los que se han desarrollado a lo largo del proyecto. Tiene como objetivo transformar el esquema de la base de datos utilizada en la aplicación actual en uno totalmente diferente, que se adapte a los modelos creados y utilizados por el nuevo sistema.

La tarea se complica aún más al observar que el esquema actual no tiene, entre otras cosas, **integridad referencial**, es decir, que cada entidad es aislada y no hay ningún tipo de referencia o *foreign key* a otras entidades. Por tanto, el *script* ha de ser capaz de corregir todo fallo posible y conseguir que toda la información que se almacene en el nuevo sistema no viole ninguna de sus restricciones.

Para conseguir este objetivo, el primer paso necesario es *parsear* el *dump* de la base de datos actual en busca de todas las inserciones de datos. Para ello se utiliza un paquete llamado **sqlparse**, que puede obtenerse con el comando `pip install sqlparse`. Éste módulo es capaz de *parsear* y separar en *tokens* el contenido de un volcado de una base de datos *MySQL*. Gracias a él será posible reconocer cada uno de las inserciones realizadas en las tablas del antiguo esquema.

Para poder convertir los datos obtenidos de las inserciones en un formato reconocido por la nueva base de datos, se han desarrollado dos clases principales **Table** y **Column**, para simular el esquema relacional utilizado en la aplicación y relacionarlo con el esquema de la antigua aplicación. De cada tabla se indicarán el nombre de la misma en el nuevo sistema, el nombre del antiguo esquema y su listado de columnas. De cada columna se especificará su nombre y, gracias a varios atributos, se podrá referenciar a una columna del antiguo esquema, así como modificar su valor. Los atributos que puede poseer una columna son los siguientes:

- `is_primary`: indica si la columna representa la clave primaria de la tabla.
- `is_sequence`: normalmente se utiliza en conjunto a `is_primary`, e indica que dicha columna representa un índice numérico autoincremental.
- `old_table_index`: hace referencia al índice de la columna equivalente en el antiguo esquema.
- `default`: permite indicar un valor por defecto para la columna en caso de no existir en el esquema original.
- `foreign_key`: se utiliza para especificar que la columna representa una referencia a otra tabla, e indica el nombre de dicha tabla y columna.

- `create_foreign_key`: permite crear una nueva fila en la tabla referenciada en caso de no encontrar la referencia solicitada.
- `transform`: permite modificar el valor original de la columna.
- `foreign_transform`: al igual que `transform`, realiza transformaciones sobre el valor original de la columna. Sin embargo, se utiliza conjuntamente a `foreign_key` para encontrar un valor modificado en la tabla referenciada.
- `format`: al igual que los atributos anteriores, realiza una transformación de la información de la columna, pero ésta no se produce hasta que todas las tablas han sido analizadas.
- `integrity`: permite modificar el valor de la columna en base a una condición generada analizando el resto de columnas de la entrada.

```
Table('users', 'usuarios', (
  Column('id', is_primary=True, is_sequence=True),
  Column(
    'user_type_id',
    old_table_index=lambda x: x[7] if x[6] == 'Otros' else x[6],
    foreign_key=('user_types', 'name'),
    create_foreign_key=lambda x: [x[7]]),
  Column('user_name', old_table_index=0),
  Column('first_name', old_table_index=2, transform=lambda x: x.title()),
  Column('last_name', old_table_index=3, transform=lambda x: x.title()),
  Column('DNI', old_table_index=4, transform=[
    lambda x: x.strip() if x else None,
    lambda x: None if not x else x]),
  Column('email', old_table_index=5),
  Column('password',
    old_table_index=1,
    transform=lambda x: generate_password_hash(x,
      method='pbkdf2:sha512:100000',
      salt_length=16)),
  Column('can_login',
    old_table_index=8,
    transform=lambda x: True if x else False),
  Column('can_write',
    old_table_index=8,
    transform=lambda x: False if x == 'Usuario' else True),
  Column('can_admin',
    old_table_index=8,
    transform=lambda x: True if x == 'Administrador' else False),
  Column('language', default='es')))
```

Figura F.1: Ejemplo de definición de la entidad **User**

Una vez definidas las entidades del nuevo esquema y su relación con las del esquema antiguo, el *script* itera sobre cada una en un orden concreto, para evitar buscar integridades referenciales en tablas aún no analizadas. En cada una de las tablas recorre todas las filas extraídas previamente y realiza los análisis y transformaciones indicados, asociando sus valores a las columnas correspondientes en el nuevo esquema. Finalmente, genera un fichero **SQL** que puede importarse en la base de datos.

Aunque este *script* se ha desarrollado teniendo en mente un esquema muy concreto, el de esta aplicación, es lo suficientemente genérico como para poder utilizarse en cualquier otra migración de bases de datos. Con añadir ciertas funcionalidades extra, este *script* podría dar lugar a un interesante proyecto completo totalmente independiente.

Para ejecutar este *script*, es necesario utilizar *Python* en su versión 3, e indicar el fichero con el *dump* original a la hora de llamarlo. Su ejecución por línea de comandos se realiza con la siguiente instrucción:
`python database_migration.py [old_schema_file] [output_sql_file]`

F.3. Migración de material

Nota: debido a su extensión, este *script* no se ha incluido en el apéndice. En caso de querer acceder a él, podrá encontrarse en el fichero *files_migration.py* incluido entre los archivos entregados del proyecto.

Con este *script* se consigue transferir todo el material adjunto digital asociado a los elementos de la aplicación existente. Para ello se aprovecha de una característica habilitada del servidor web que, en ciertos ámbitos, podría considerarse un fallo de seguridad: la posibilidad de listar todos los ficheros almacenados en la aplicación desde el propio servidor sin necesidad siquiera de estar autenticado en el sistema.

El *script* realiza una petición web a una *URL* que devuelve el listado de ficheros de un directorio, para posteriormente parsear el documento *HTML* que devuelve el servidor. Desde ahí, extrae las *URL* de cada uno de los ficheros, y lo descarga a un directorio del sistema. Además, en caso de ficheros comprimidos en formato **ZIP**, se encarga de analizarlos e intentar descomprimirlos. Finalmente, crea un *script* **SQL** para con las consultas necesarias para insertar en la base de datos las filas necesarias para que la nueva aplicación reconozca la existencia de los ficheros descargados.

A la hora de analizar los ficheros e intentar detectar su formato, el *script* toma un camino alternativo al habitual, basado en comprobar la extensión del fichero para indicar su tipo. En su lugar, utiliza una característica presente en la mayoría de tipos de ficheros, su **magic number**: los primeros *bytes* de cualquier fichero indican formato. Por ejemplo, un fichero **PDF** comenzará con los siguientes *bytes*: `25h 50h 44h 46h`.

En caso de encontrar un fichero comprimido, se siguen diferentes métodos de actuación, dependiendo de varias circunstancias. Los únicos ficheros que se descomprimen son los ficheros en formato *ZIP*, ya que es el único formato que es posible leer de forma nativa

en *Python*. Se ha decidido no descomprimir los formatos *RAR* y *7z* ya que requieren de herramientas externas que no funcionan correctamente en caso de encontrar archivos con características concretas. Aún así, no todos los archivos *ZIP* se descomprimen: en caso de encontrar una estructura muy compleja dentro del archivo, con más de 15 ficheros o con una estructura de subdirectorios anidados, se omite la descompresión, por asumir que el fichero puede contener un *software* o una imagen de *CD* o *DVD*.

Todos los archivos que genere el *script* se irán almacenando en el directorio utilizado por la aplicación para guardar el material digital asociado a los elementos existentes en la misma. Una vez se hayan guardado todos los ficheros existentes, el *script* creará un fichero **SQL** con la información a añadir en la base de datos para que la aplicación reconozca la existencia de estos ficheros, así como su nombre, su ubicación en el sistema de archivos y su formato **MIME**[24].

```
def should_uncompress_file(file):
  num_dirs = 0
  num_nested_dirs = 0
  num_files = 0
  for x in file.infolist():
    if x.file_size == 0:
      slash = '/' if x.filename[-1] == '/' else False
      backslash = '\\' if x.filename[-1] == '\\' else False
      if slash or backslash:
        num_dirs += 1
        if x.filename.index(slash or backslash) + 1 != len(x.filename):
          num_nested_dirs += 1
    else:
      num_files += 1
  if not num_dirs or not num_nested_dirs:
    return num_files <= 15
  return False
```

Figura F.2: Análisis de estructuras complejas de archivos en ficheros *zip*

Importante: el fichero *SQL* generado por este *script* no ha de importarse a la base de datos sin previamente realizar la migración de datos del anterior apéndice. En caso contrario, la ejecución del *script* fallará y la aplicación no reconocerá la existencia de los ficheros almacenados en la misma. Además, es necesario tener acceso a la dirección *IP* de la anterior plataforma. De lo contrario no sera posible obtener el listado de ficheros para su descarga.

Es necesario utilizar *Python 3* para ejecutar el *script*, cuya ejecución se realiza desde la línea de comandos utilizando la siguiente instrucción:
`python files_migration.py [output_directory_path] [output_sql_file]`

[24] cadenas de texto utilizadas en el intercambio de archivos en Internet, para representar el formato del archivo contenido.

G. Scripts propios de la aplicación

G.1. Fichero de configuración del protocolo *wsgi*

Nota: el fichero se encuentra en `/var/www/photonics/uswgi.ini` y es utilizado por el servicio `photonics` en el arranque, creando el *socket* de comunicación con *Nginx*.

```
[uwsgi]
# application root dir
base = /var/www/photonics/htdocs/
# python module loaded
module = app
# flask app object
callable = application
# virtualenv root dir
home = %(base)../env/
pythonpath = %(base)
chdir = %(base)
# process setuid
uid = photonics
# process setgid
gid = nginx
# master process
master = true
# number of workers
processes = 4
# unix socket file
socket = /var/www/photonics/%n.sock
# socket owner
chown-socket = photonics:nginx
# socket permissions
chmod-socket = 660
# spawn new processes
close-on-exec = true
close-on-exec2 = true
# no file size limit
limit-post = 0
socket-timeout = 86400
http-timeout = 86400
# exit on SIGTERM
die-on-term = true
# clean generated files
vacuum = true
# log file
logto-syslog = photonics
```

G.2. Abstracción sobre *Flask*

Nota: únicamente se va a proceder a explicar el funcionamiento básico del *script* debido a su amplia extensión. Para ver el código fuente, es necesario acceder al fichero `photon.py` situado en la raíz de la aplicación, `/var/www/photonics/htdocs/`.

Gracias al *framework* **Flask**, la complejidad asociada al desarrollo de una aplicación se reduce notablemente, ya que incluye de serie un gran número de herramientas y funcionalidades que abstraen al programador del complejo sistema requerido para analizar peticiones web y servir los resultados al usuario.

A pesar de ello, y debido a su carácter genérico, no es capaz de evitar ciertas tareas repetitivas de código, como puede ser el control de la autenticación de usuarios, y la inyección de un conjunto de datos similares en todas las vistas. Para solucionarlo, se ha implementado una capa por encima de *Flask*, que a su vez actúa de *pseudo-framework*, abstrayendo al desarrollador de la mayoría de las peculiaridades y herramientas de *Flask*. Por otro lado, también se encarga de configurar los parámetros y opciones básicas requeridas por el *framework* para su correcto funcionamiento, como son la incorporación de los diferentes módulos y *plugins* utilizados a lo largo de la aplicación.

Una de sus utilidades principales es la definición de controladores, permitiendo especificar el listado de métodos que pueden aceptar (*GET* o *POST*) o las permisos de acceso que requieren. También permite especificar la *URL* a la que responderá el controlador, aunque se ha diseñado un sistema por el que es posible generar esta *URL* de forma automática, basándose en el nombre del controlador.

```
def _get_route_path(name):
  name = '/' + name.replace('_', '/')
  return name
def _add_route_query(name, query):
  name += '/' + '/'.join('<' + x + '>' for x in query) if query else ''
  return name
def _create_route(func, route, methods):
  _app.add_url_rule(route, None, func, methods=methods)
def route(methods=['GET'], query=(), route=None):
  def decorator(func):
    path = _get_route_path(func.__name__) if not route else route
    path = _add_route_query(path, query)
    _create_route(func, path, methods)
    return func
  return decorator
```

Figura G.1: Generación de rutas basándose en el nombre de los controladores.

Su siguiente función principal es la gestión de usuarios. A través del *decorator* definido en la figura **8.13** es posible controlar los permisos de acceso a un controlador. Es

posible restringir el acceso a una ruta sólo a usuarios anónimos, limitado en este caso a las rutas de *login* y recuperación de contraseña. En caso de forzar la autenticación del usuario para acceder a una ruta, es posible comprobar si éste posee permisos de escritura o administración para acceder al controlador deseado.

La tercera utilidad crítica de este archivo es la inyección de variables en las vistas. Desde un controlador es posible enviar datos específicos a la vista para ser renderizados. Sin embargo, existen un conjunto de datos accesibles por todas las vistas, como puede ser el listado de categorías. Sería una tarea muy tediosa y propensa a fallos el incluir estas variables en todos y cada uno de los controladores. Para ello, se ha sustituido el método `render_template` de *Flask*, encargado de renderizar una vista, para inyectar siempre ciertos datos, los cuales dependen de si el usuario activo está autenticado o no en el sistema.

```
def render_anonymous_template(template, **kwargs):
  if 'notification' in kwargs:
    notification = models.app.Notification.query.get(kwargs['notification'])
    if notification:
      kwargs['notification'] = notification.value
    else:
      kwargs['notification'] = None
  return render_template(template, **kwargs)
```

Figura G.2: Ejemplo de renderizado de vistas para usuarios anónimos

Finalmente, también se encarga de inyectar automáticamente todos los parámetros de configuración de la aplicación definidos en el fichero `config.py`, permitiendo acceder por completo a la clase `AppConfig` de este fichero para extraer datos como el listado de idiomas aceptados o las rutas y directorios donde se almacenan los diferentes archivos asociados a entradas.

```
@_app.context_processor
def context_processor():
  return {key: value
    for key, value in vars(config.AppConfig).items()
    if not callable(value) and not key.startswith('__')}
```

Figura G.3: Inyección de parámetros de configuración en las vistas

G.3. Boilerplate de la aplicación

G.3.1. Modelos

```
class Entity(db.Model):
  __tablename__ = 'tablename'
  # Complex constraints
  __table_args__ = (db.UniqueConstraint('foreign_id', 'name'),)
  id = db.Column(db.Integer, primary_key=True)
  foreign_id = db.Column(db.Integer, db.ForeignKey('tablename.id'))
  foreign_column = db.relationship('Entity')
  name = db.Column(db.String(64), nullable=False)
  @hybrid_property
  def derived_property(self):
    if not self.foreign_id
      return 0
    else:
      return self.foreign_id
# Indexes with functions or partial indexes
db.Index('index', func.lower(Entity.name), unique=True)
db.Index(
  'partial_index',
  func.lower(Entity.name),
  postgresql_where=Entity.foreign_id == None,
  unique=True)
```

G.3.2. Controladores

```
@photon.route(methods=('GET', 'POST'), query=params, route=url)
@photon.authenticated(anonymous=True|False, write=[T|F], admin=[T|F])
def route_name(params):
  if request.method == 'GET':
    pass
  # controller logic
  ...
  return photon.render_logged_template(
    'view_file.html', # view template
    view_var_1=value, # data sent to the view
    view_var_2=value,
    ...)
```

G.3.3. Formularios

```
class CreateUser(Form):
  user_name = StringField('user_name', validators=[
```

```
    DataRequired(),
    forms.validators.CustomLength(max=64),
    Regexp('^[a-zA-Z0-9]+$')])
  first_name = StringField('first_name', validators=[
    DataRequired(),
    forms.validators.CustomLength(max=128)])
```

G.3.4. Vistas

Una vista es un fichero *HTML* totalmente normal. Sin embargo, existen ciertas cadenas de caracteres que permiten realizar acciones, como renderizar una variable o ejecutar instrucciones de código.

Para mostrar el valor de una variable o una función se utiliza el comando {{ `var_name|e` }}. Los caracteres `|e` permiten sanear su valor, evitando así ataques *XSS*. Las funciones, o macros, se encuentran definidas en el fichero `macros.html`, y permiten generar estructuras *HTML* reutilizables en varias zonas de la aplicación.

Es posible también ejecutar ciertas estructuras de control de flujo, como **if** o **for**:

```
{% if condition %}
  ...
{% elif condition %}
  ...
{% else %} ... {% endif %}
{% for x in list %} ... {% endfor %}
```

Figura G.4: Estructuras de control de flujo en las vistas

Es importante tener en cuenta que todas las vistas renderizan el núcleo central de la aplicación, mientras que las barras de menú laterales y superiores se generan en una vista maestra, `layout.html`. Es necesario indicar al sistema dicha vista maestra y que partes de la misma se van a completar, por lo que el esqueleto básico de la vista tendrá la siguiente forma:

```
{% extends layout.html %}
{% block scripts %}
  <!-- scripts -->
{% endblock %}
{% block body %}
  <!-- view content -->
{% endblock %}
{% block modals %}
  <!-- popup modals -->
{% endblock %}
```

Figura G.5: Esqueleto de código de todas las vistas

G.4. Scripts de búsqueda

G.4.1. Búsqueda simple

```
search_query = search_query.strip()
entries = OrderedSet()
if search_query.isdigit():
  entries.update(Entry.query
    .filter(cast(Entry.formatted_id, String).like('%' + search_query + '%'))
    .order_by(Entry.formatted_id)
    .all())
if len(search_query) >= 3 and len(search_query.split()) == 1:
  entries.update(Entry.query
    .filter(Entry.serial_number.ilike('%' + escape_like(search_query) + '%'))
    .all())
entries.update(Entry.query.search(
  search_query,
  vector=Entry.name_search_vector, sort=True).all())
entries.update(Entry.query.search(
  search_query,
  vector=Entry.description_search_vector, sort=True).all())
entries.update(Entry.query.search(
  search_query,
  vector=Entry.comments_search_vector, sort=True).all())
entries.update(Entry.query.search(
  search_query,
  vector=Entry.full_search_vector, sort=True).all())
```

G.4.2. Búsqueda avanzada

G.4.2.1. Petición POST

```
@photon.route(methods=('POST',), route='/inventory/advanced_search')
@photon.authenticated()
def inventory_advanced_search():
  form = forms.adv_search.AdvancedSearch.create_instance()
  return redirect(
    url_for('inventory_advanced_search_query',
    search_query=form.get_query()))
```

G.4.2.2. Serialización de la búsqueda

```
def get_query(self):
  fields = [
    self.name.data,
```

```
    get_field_id(self.category.data),
    get_field_id(self.status.data),
    self.serial_number.data,
    get_field_id(self.in_charge.data),
    get_field_id(self.documentation.data),
    self.description.data,
    self.comments.data,
    get_field_id(self.actual_use_user.data),
    get_field_id(self.actual_use_type.data)]
  extra_fields = {}
  for x in self.dynamic_fields.keys():
    field = getattr(self, x).data
    if field:
      extra_fields[x[len('field_'):]] = field
  fields.append(extra_fields)
  extra_fields = {}
  for x in self.category_fields.keys():
    field = getattr(self, x).data
    if field:
      extra_fields[x[len('category_field_'):]] = field
  fields.append(extra_fields)
  return utils.base64_encode(json.dumps(fields))
```

G.4.2.3. Deserialización de la búsqueda

```
def set_query(self, query):
  fields = json.loads(utils.base64_decode(query))
  self.name.data = fields[0]
  self.category.data = get_field_from_id(Category, fields[1])
  self.status.data = get_field_from_id(EntryStatusType, fields[2])
  self.serial_number.data = fields[3]
  self.in_charge.data = get_field_from_id(User, fields[4])
  self.documentation.data = get_field_from_id(EntryDocumentationType, fields[5])
  self.description.data = fields[6]
  self.comments.data = fields[7]
  self.actual_use_user.data = get_field_from_id(User, fields[8])
  self.actual_use_type.data = get_field_from_id(EntryUseType, fields[9])
  for x in fields[10].keys():
    field_name = 'field_' + x
    if hasattr(self, field_name):
      field = getattr(self, field_name)
      field.data = fields[10][x]
  for x in fields[11].keys():
    field_name = 'category_field_' + x
    if hasattr(self, field_name):
```

```
      field = getattr(self, field_name)
      field.data = fields[11][x]
```

G.4.2.4. Petición GET

```
@photon.route(methods=('GET',),
  route='/inventory/advanced_search/<path:search_query>')
@photon.authenticated()
def inventory_advanced_search_query(search_query):
  form = forms.adv_search.AdvancedSearch.create_instance()
  form.set_query(search_query)
  if form.validate():
    entries = form.search()
  else:
    entries = []
    errors = utils.reduce_form_errors(form)
    if not errors:
      flash(gettext('invalid_advanced_search_form'), 'inventory')
    for error in errors:
      flash(error, 'inventory')
  return photon.render_logged_template(
    'inventory.html',
    main_title=gettext('advanced_search'),
    notification='home_message',
    collapsable_entries=True,
    entries=entries,
    adv_search_form=form,
    delete_form=None)
```

upna

Ingeniería Técnica en Informática de Gestión
Escuela Técnica Superior de Ingenieros Industriales y de Telecomunicación

XABIER-CERNIN NAPAL RONCAL

MEJORA E IMPLEMENTACIÓN DE UN SISTEMA DE GESTIÓN DE INVENTARIO PARA EL LABORATORIO DE FOTÓNICA DE LA UNIVERSIDAD PÚBLICA DE NAVARRA

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

ÍNDICE

OBJETIVOS

- Gestión de inventario
  - Categorización multinivel
  - Datos dinámicos
  - Sistema de búsquedas
- Aplicación web
  - Multiplataforma
  - Interfaz *responsive*
- Autenticación y privilegios
- Seguridad e integridad

TECNOLOGÍA – SERVIDOR

- **Máquina virtual**
  Proxmox – KVM
- **Sistema Operativo**
  CentOS 7
- **Lenguaje de Programación**
  Python 3.5
- **Motor de Bases de Datos**
  PostgreSQL 9.5
- **Servidor Web**
  Nginx 1.10

TECNOLOGÍA – FRAMEWORK

- **Back-end**
  Flask
  - Flask-Login
  - SqlAlchemy
  - WTForms
  - Babel
- **Front-end**
  Bootstrap
  - Gentelella
  - ¿jQuery?

Bootstrap

SEGURIDAD

- **Password hardening**
  PBKDF2

- **SQL Injection**
  SqlAlchemy

- **Cross-Site Scripting**
  Jinja2

- **Cross-Site Request Forgery**
  WTForms

- **¿HTTPS?**

DESARROLLO

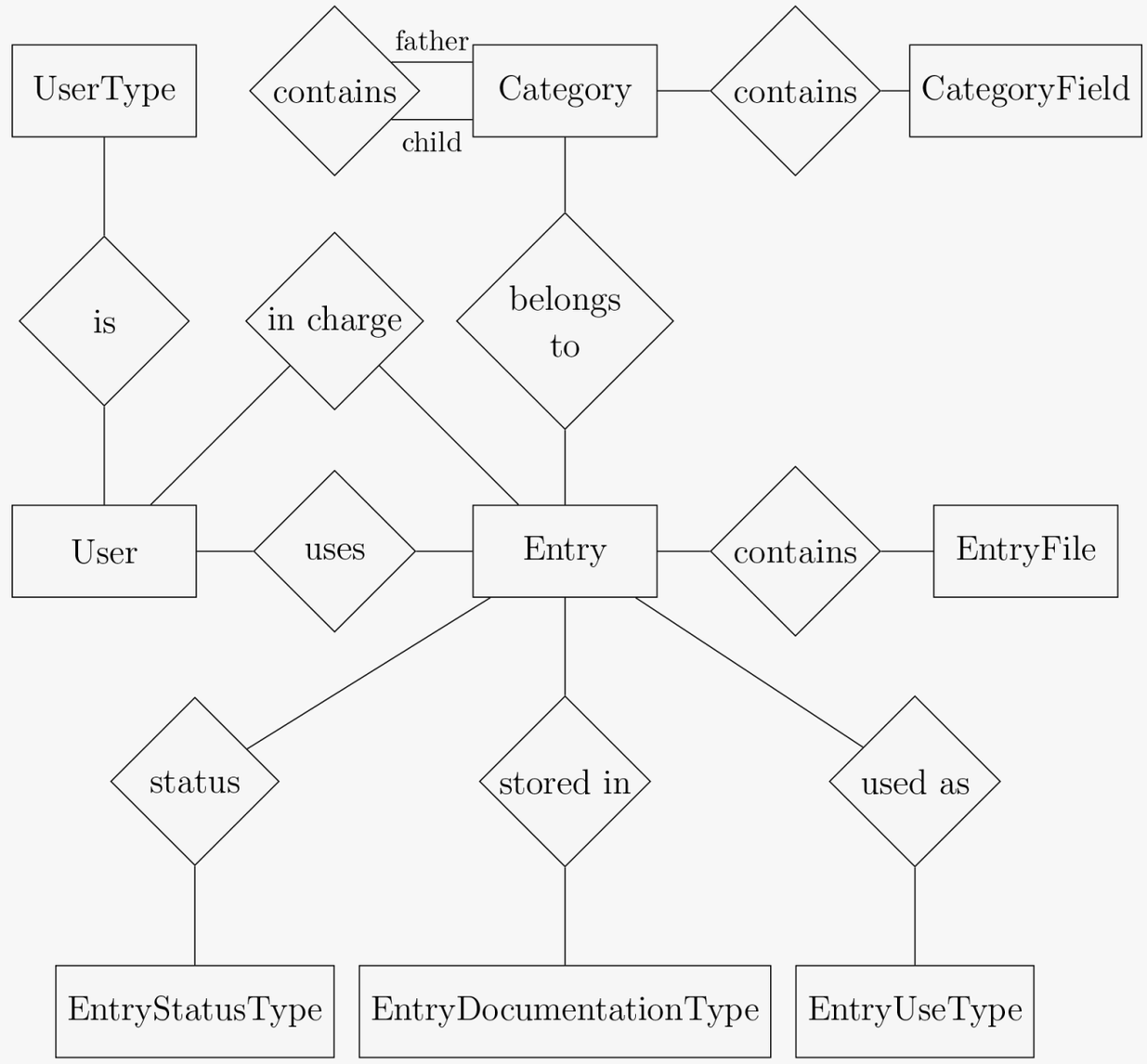
- Instalación
- Modelado y migración de datos
- Desarrollo de la aplicación
- Sistemas de búsqueda
- Panel de administración
- Copias de seguridad
- Actualizaciones del sistema

DESARROLLO – INSTALACIÓN

- Script de automatización
- Actualizaciones del sistema
- Software necesario
  Python, PostgreSQL, Nginx, FTP
- **Módulos Python**
  Flask, SqlAlchemy, WTForms, Babel, uWSGI
- **Seguridad**
  SELinux, Firewall

DESARROLLO – BASE DE DATOS

- **Modelo entidad-relación**
- **Datos dinámicos**
  PostgreSQL: *JSON*
- **¿Paso a tablas?**
  SqlAlchemy: *ORM*
- Migración de datos
- Migración de archivos

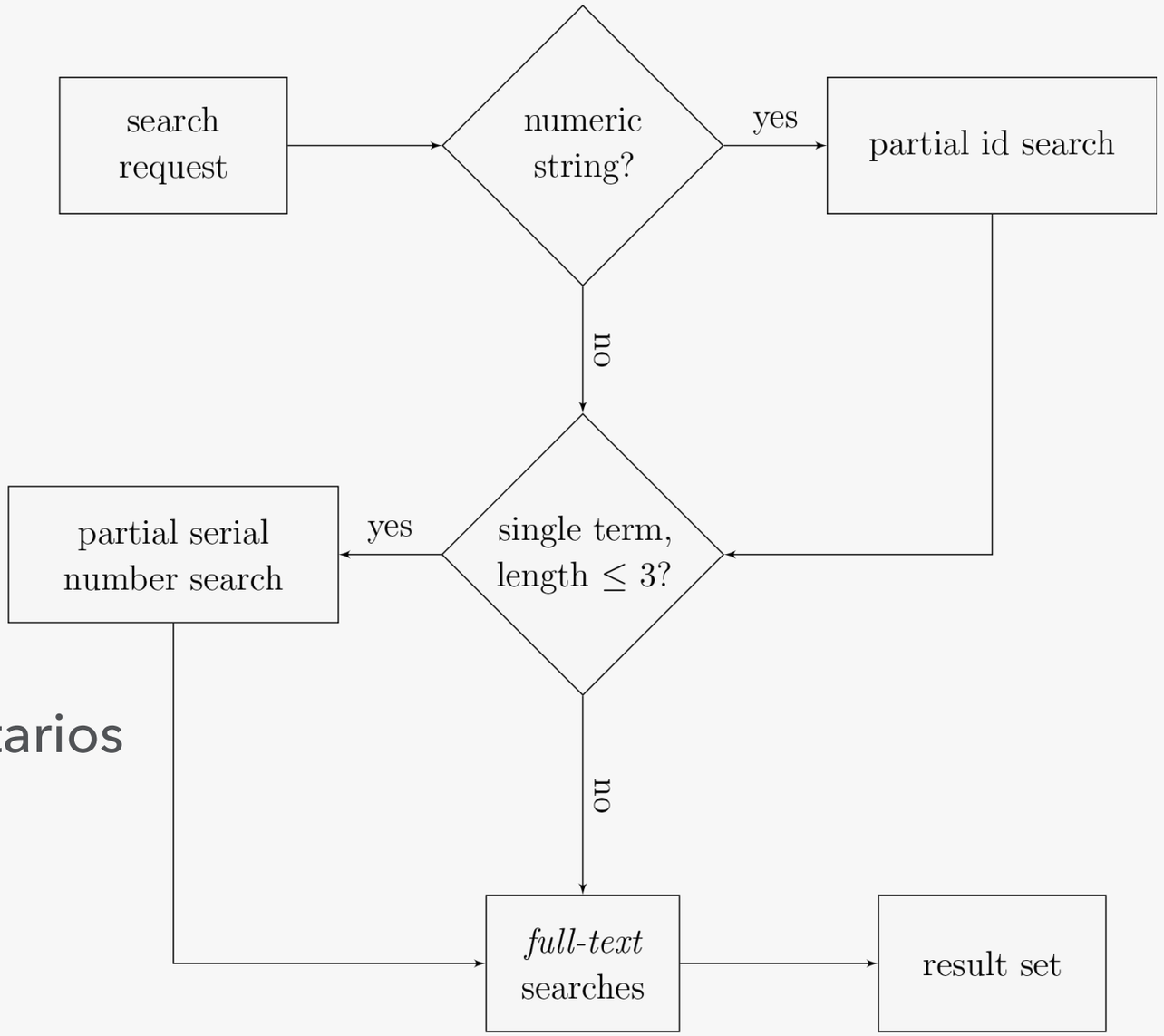


DESARROLLO – APLICACIÓN

- Autenticación y privilegios
- Recuperación de contraseña
- Internacionalización
- Gestión de inventario
  - Listado de categorías
  - Desglose de elementos
  - Inserción y edición de elementos

DESARROLLO – BÚSQUEDA

- **Búsqueda semántica**
  PostgreSQL: *full-text*
- Búsqueda simple
  - Identificador único
  - Número de serie
  - Nombre, descripción, comentarios
- Búsqueda avanzada
  - Filtro por todos los campos
  - Incluye campos dinámicos



Búsqueda simple

DESARROLLO – ADMINISTRACIÓN

- Control de la aplicación
- Inventario
  - **Árbol de categorías:** múltiples niveles
  - **Campos dinámicos:** globales y de categoría
  - **Entradas:** tipo de estado, uso y documentación
- Gestión de usuarios
- Creación de notificaciones
- Backups y actualizaciones

DESARROLLO – BACKUPS

- Dos tipos de backup
  - **Base de datos:** información e imágenes
  - **Ficheros almacenados:** material digital asociado
- Backup manual: panel de administración
  - Creación
  - Restauración
- Backup automático: creación periódica
  `http://fotonica.unavarra.es/admin/backup/auto/<user>:<password>`

DESARROLLO – ACTUALIZACIONES

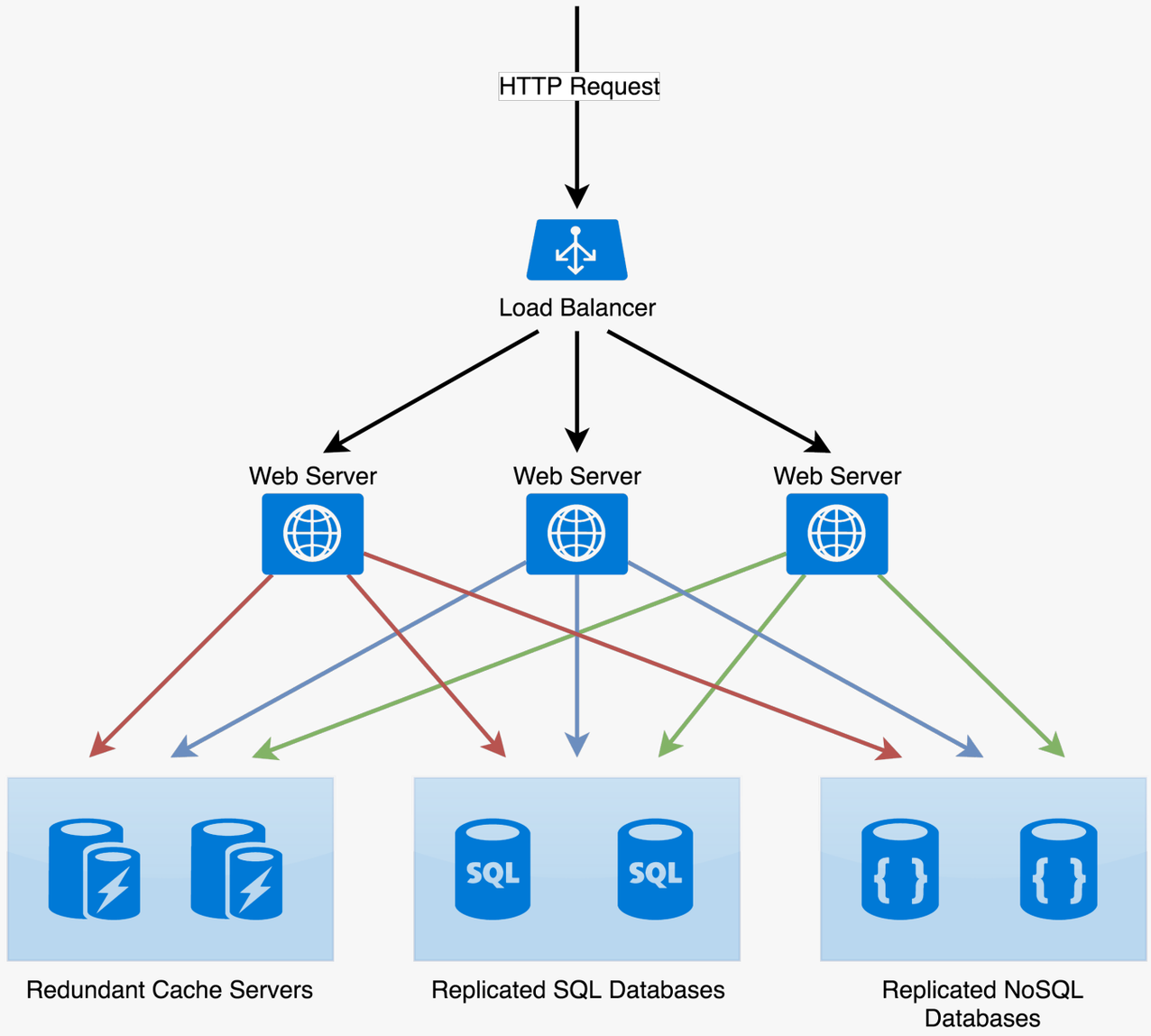
- Actualización manual desde web
- Herramienta de sistema **yum**

Precaución: posibilidad de fallos

¡Siempre realizar copia de la máquina virtual!

LÍNEAS FUTURAS

- **Generalización**
  Múltiples subelementos
- **Expansión**
  Diferentes almacenes
- **Escalabilidad**
  Instancias redundantes
- **Especialización**
  Software específico: caché, NoSQL, búsquedas
  Memcached, *MongoDB*, *Sphinx*



Demostración:

http://fotonica.unavarra.es

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa